PLACES: PARALLELISM FOR RACKET

by

Kevin B. Tew

A dissertation submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

School of Computing

The University of Utah

August 2013

THE UNIVERSITY OF UTAH GRADUATE SCHOOL

STATEMENT OF DISSERTATION APPROVAL

The dissertation of Kevin B. Tew
has been approved by the following supervisory committee members:

Matthew Flatt, Chair	March 17, 2013
	Date Approved
John Regehr, Member	March 5, 2013
	Date Approved
Mary Hall, Member	March 26, 2013
	Date Approved
Matthew Might, Member	March 6, 2013
	Date Approved
Peter Dinda, Member	
	Date Approved

and by Alan Davis, Chair of the School of Computing
and by Donna M. White, Interim Dean of The Graduate School

ABSTRACT

Places and distributed places bring new support for message-passing parallelism to Racket. This dissertation describes the programming model and how Racket's sequential runtime-system was modified to support places and distributed places. The freedom to design the places programming model helped make the implementation tractable; specifically, the conventional pain of adding just the right amount of locking to a big, legacy runtime system was avoided. The dissertation presents an evaluation of the places design that includes both real-world applications and standard parallel benchmarks. Distributed places are introduced as a language extension of the places design and architecture. The distributed places extension augments places with the features of remote process launch, remote place invocation, and distributed message passing. Distributed places provide a foundation for constructing higher-level distributed frameworks. Example implementations of RPC, MPI, map reduce, and nested data parallelism demonstrate the extensibility of the distributed places API.

For Cheryl, Tanner, and Kanani

CONTENTS

LIST OF FIGURES

ACKNOWLEDGMENTS

I am greatful to my advisor Matthew Flatt for his patience in guiding me through the Ph.D. process. Matthew always had a positive suggestion for me when I was lost or discouraged. Much thanks goes to all my committee members: John Regehr, Mary Hall, Matthew Might, and Peter Dinda, whose critiques and suggestions improved my work.

CHAPTER 1

INTRODUCTION

1.1 Statement of Problem

Software developers and academic researchers need a cost-effective and time-efficient means to prototype, design, and experiment with parallel programs. Ideally, such a playground would consist of a multicore workstation and a dynamic programming language such as Racket, Python, or Ruby.

Unfortunately, dynamic language runtimes evolved into their present form without forethought or design for supporting parallelism. Dynamic languages have attempted to obtain parallelism by adding OS scheduled threads and locks to their respective language runtimes. Language interpreters, however, are some of the most complex types of software. The lifecycle and structure of data in interpreters are dynamic and difficult to predict. Objects created in one portion of the interpreter often travel throughout the interpreter and have highly varied lifetimes. These lifecycle characteristics lead to huge numbers of invariants that make locking a multithreaded interpreter extremely difficult. As languages adapt and evolve, each change to the runtime requires language implementers to check and reverify that the invariants of their locking designs are preserved.

Ensuring locking correctness has proved too difficult, so implementations have surrendered parallelization attempts by resorting to big interpreter locks (GIL), which serialize execution of OS level threads in the language runtime. [5, 34, 42, 46].

1.2 Thesis Statement

Existing dynamic-language virtual machines can be transformed into effective parallel and distributed computing platforms by adding core parallel primitives and using the abstraction power of programming languages to incorporate and extend those parallel primitives into higher-level parallel constructs.

1.3 Context of Work

Parallel programs target either shared-memory or distributed-memory architectures. Both styles of parallelism, shared-memory and distributed-memory, are typically implemented as libraries and

extensions of languages such as C, C++, and Fortran. Unfortunately, these languages and parallel libraries demand a lot of expertise from the programmer to produce results.

Shared-memory systems require the use of threads and synchronization primitives to implement parallelism. In shared-memory programs, programmers are left solely responsible for data consistency across different contexts of execution. Shared-memory systems provide synchronization primitives, but programmers have to ensure that synchronization is used correctly and is present in every location where it is needed for safety. Just as there is always one more memory leak to plug in a C program, there is always one more synchronization primitive to add to a shared-memory program. Supposing all necessary synchronization is present in a program and composed correctly to avoid deadlock, such synchronization correct programs often spend significant portions of their time in synchronization primitives instead of accomplishing meaningful, parallel work.

Distributed-memory programs typically employ message-passing libraries, exemplified by MPI, to communicate between compute nodes. Message-passing systems avoid the shared mutation problem of shared-memory systems by copying data and sending copies to other contexts of execution. The downside to distributed-memory systems is the cost of data serialization and data transport between different contexts of execution. Message-passing programmers must also hurdle the obstacles of remote process launch and authentication over a cluster of machines before beginning parallel application development.

Distributed-memory techniques scale to huge numbers of systems and processors, well beyond the size of the largest shared-memory machines. Because distributed-memory programmers are forced to address data placement and communication costs from the earliest stages in their programs' lifecycles, distributed-memory programs typically translate easily to shared-memory systems. Translated distributed-memory programs usually perform well on shared-memory machines because they naturally avoid cache and memory contention and minimize communication. The corollary for shared-memory systems is not true; generally they do not scale to massive processor counts nor are they easily translated to distributed-memory programs.

1.3.1 Method

This dissertation demonstrates the addition of parallelism to the Racket language virtual machine through the implementation of places and distributed places. Racket is a functional programming language descended from LISP and Scheme. Only the name "places" is borrowed from the X10 [10] language project. The design and implementation of Racket places is distinct and unique from X10 places.

Places implements shared-nothing, message passing parallelism. The process-like memory isolation provided by places is enforced by the language runtime. Places, however, are implemented as shared-memory, operating system threads. This design choice allows the runtime implementer to exploit shared memory to increase communication performance and minimize resource utilization. Finally, the places abstraction emphasizes locality and communication costs, the true bottlenecks to parallelization.

Places implementation and development occured in a series of steps. First, the Racket VM was modified to allow multiple racket interpreters to execute in parallel. An abstraction for creating OS schedulable threads was created. Then, mutable global variables had to be converted to place local variables.

The newly, place-retrofitted virtual machine requires a hierarchical, parallel garbage collector, allowing individual place instances to collect independently. Separate garbage collection realms gives each place the independence necessary to achieve maximum parallelism. A small, global shared heap spans across all places, facilitating place creation and interplace communications. In the future, it is expected that large amounts of static content will be shared across places. Some examples of static content that could be shared include bytecode, jitted code, static data tables, and module definitions.

Places communicate over bidirectional place channels. Place channels maintain separation of places by ensuring that message contents are copied from one place to another. Passing an arbitrary reference between places is strictly prohibited.

The distributed computing framework, *distributed places*, allows distributed computing among a network of machines. Distributed places channels communicate over TCP sockets. New distributed places can be spawned during the execution of a distributed program. Racket's distributed programming environment provides facilities for detection of remote process failure.

The programming model for places and distributed places is a shared-nothing message passing model. The places implementation, however, carves out an exception for certain algorithms and problem sizes that benefit from shared-memory communication. Places accommodates a subset of such algorithms through the use of shared vectors. Shared-vector primitives permit a restricted form of shared-memory data structures while preserving the integrity of the language virtual machine.

Distributed places strictly adheres to the shared-nothing principle. Distributed places do not provide any type of global address space across VM instances in a distributed system. The user is responsible to distribute data to remote compute nodes through distributed place channels or other means external to the distributed places framework.

The choice of a shared-nothing programming model is central to the places design and implementation. The isolation of the shared-nothing model allows a sequential language runtime, such as Racket, to be converted to a parallel language runtime without the need to completely rewrite the runtime from scratch. The shared-nothing programming model also enables the conversion of the language runtime conversion without the extensive use of locks.

1.4 Contributions

The core idea of the dissertation is that an effective parallel runtime can be grown from a sequential language runtime by using process-like isolation instead of locks. The idea of isolation extends into the garbage collector as well. The isolation of place-specific garbage collectors allows individual places to collect independently of each other.

There are algorithms that are better suited to shared memory environments than shared-nothing environments. The idea of limited, yet shared vectors permits the use of shared-memory vectors while preserving the integrity of the language virtual machine.

Last, the place primitives provide a foundation for building further parallelism constructs, such as distributed places. Distributed places extend place channels by coupling them to TCP sockets. These distributed place channels allow place messages to flow from one machine to another.

CHAPTER 2

RELATED WORK

Considerable prior work addresses the problem of language design for parallelism. In contrast to most other work, this dissertation specifically addresses the problem of converting an existing sequential run-time system into a parallel runtime system. Racket's characteristics as a dynamic, functional language with mutation presents unique challenges for parallelization. The Places design builds upon the related work below and shows that parallelism can be added to an existing language virtual machine such as Racket.

2.1 Languages with Parallelism

2.1.1 Racket's futures

Racket's futures [45], like places, provide a way to add parallelism to a legacy runtime system. Futures are generally easier to implement than places, but the programming model is also more constrained. Specifically, a place can run arbitrary Racket code, but a future can only run code that is already in the "fast path" of the runtime system's implementation. There are, however, a few situations where futures are less constrained, namely when operating on shared, mutable tree data structures. Some tasks (including many of the benchmarks in Section 3.7), are well-supported by both futures and places and, in those cases, the performance is almost identical. It is expected that the development of new constructs for parallelism in Racket will internally combine futures and places to get the advantages of each.

2.1.2 Concurrent Caml Light

Concurrent Caml Light [14] relies on compile time distinction between mutable and immutable objects to enable thread local collection. Concurrent Caml Light gives its threads their own nurseries, but the threads all share a global heap. Concurrent Caml Light is more restrictive than Racket places. In Concurrent Caml Light, only immutable objects can be allocated from thread-local nurseries; mutable objects must be allocated directly from the shared heap. Concurrent Caml

Light presumes that allocation of mutable objects is infrequent and that mutable objects have longer life spans. While Racket takes inspiration from Caml Light's garbage collector, Racket's garbage collector performs the same regardless of mutable object allocation frequency or life span.

2.1.3 Erlang

Erlang [40] from the language VM's perspective is a pure functional language without destructive update. Erlang uses a hybrid memory management scheme similar to Racket's master and place local GC realms. A pointer directionality invariant, stating that pointers only point from local heaps to the shared message heap, permits independent collection of local processes. Racket uses the same type of pointer directionality invariant to allow independent place-local garbage collection.

All Erlang message contents have to be allocated from the shared heap. This allows O(1) message passing, assuming message contents are correctly allocated from the shared heap, not from the Erlang process' local nursery. Erlang employs static analysis to try to determine which allocations will eventually flow to a message send. Since messages are always allocated to the shared heap, Erlang must collect the share heap more often then Racket, which always allocates messages into the destination place's local heap. Erlang's typical programming model has many more processes than CPU cores and extensive message exchange. Places programming model is normally one place per CPU core, with less message passing traffic.

2.1.4 Haskell

Haskell [28, 29] is a pure functional language with support for concurrency. Currently, Haskell garbage collection is global; all threads must synchronize in order to garbage collect. The Haskell implementors plan to develop local collection on private heaps, exploiting the predominance of immutable objects similarly to Concurrent Caml Light's implementation. In contrast to pure functional languages, Racket programs often include mutable objects, so isolation of local heaps, not inherent immutability, enables a place in Racket to independently garbage-collect a private heap.

2.1.5 Manticore

Manticore [19] from its beginning, was designed to be a parallel language. Like Erlang and Haskell, the Manticore programming language (PML), does not have mutable datatypes. In contrast, Racket places adds parallelism to an existing sequential language with mutable datatypes. As places matures, the plan is to add multilevel parallelism, similar to Manticore.

2.1.6 Matlab

Matlab provides programmers with several parallelism strategies. First, compute intensive functions, such as BLAS matrix operations, are implemented using multithreaded libraries. Simple Matlab loops can be automatically parallelized by replacing `for` with `parfor`. Matlab's automatic parallelization can handle reductions such as min, max, and sum, but it does not parallelize loop dependence. However, it should be noted that slice operations are more central to the Matlab programming model than for loops. Matlab also provides task execution on remote Matlab instances and MPI functionality. Rather than adding parallelism through libraries and extensions, places integrates parallelism into the core of the Racket runtime.

2.1.7 Python's multiprocessing library

Python's multiprocessing library [35] provides parallelism by forking new processes, each of which has a copy of the parent's state at the time of the fork. In contrast, a Racket place is conceptually a pristine instance of the virtual machine, where the only state a place receives from its creator is its starting module and a communication channel. More generally, however, Python's multiprocessing library and Racket's places both add parallelism to a dynamic language without retrofitting the language with threads and locks.

Communication between Python processes occurs primarily through OS pipes. The multiprocessing library includes a shared-queue implementation, which is implemented by using a worker thread to send messages over pipes to the recipient process. Any "picklable" (serializable) python object can be sent through a multiprocessing pipe or queue. Python's multiprocessing library also provides shared-memory regions implemented via `mmap()`. Python's pipes, queues and shared-memory regions must be allocated prior to forking children, which need to use them. Racket's approach offers more flexibility in communication; channels and shared-memory vectors can be created and sent over channels to already-created places; channels can communicate immutable data without the need for serialization.

2.1.8 Python and Ruby

Python and Ruby implementors, like Racket implementors, have tried and abandoned attempts to support OS-scheduled threads with shared data [5, 34, 42]. All of these languages were implemented on the assumption of a single OS thread—which was a sensible choice for simplicity and performance throughout the 1990s and early 2000s—and adding all of the locks needed to support OS-thread concurrency seems prohibitively difficult. A design like places could be the right approach for those languages, too.

Alterntative Ruby implementations such as Rubinius2 and JRuby provide parallelism by mapping Ruby threads to OS threads. JRuby and Rubinius are complete reimplementations of the Ruby VM from the ground up. These shared-memory threads implementations of Ruby are vulnerable to subtle race conditions due to opaque sharing and caching in their virtual machines implementations. As with all shared-memory parallelism implementations, parallel ruby programmers are responsible for using locks to ensure data consistency.

2.1.9 Perl

Perl [7, 6] supports thread based parallelism through interpreter threads. Although many distributions build Perl with interpreter threads enabled, threads are not enabled by default. Anecdotal evidence indicates that disabling threads can result in a 10% to 20% increase in performance. Each creation of a thread spawns a new interpreter much like places. Interthread communication is achieved using the shared-memory scalars, arrays, and hashs that are explicitly annotated as shared. All other variables are implicitly thread local. Shared data can only store scalars or references to other shared variables. Shared variables have slightly different semantics than thread local variables. Some perl operations, such as changing array length via `$#array` and autovivification, do not work when used with shared variables. Unlike places, Perl threads provide the traditional locks and condition variables for maintaining data isolation of shared variables. In contrast, places relies on message passing instead of shared memory for communication.

In the previous Perl threading model, 5005threads [11], all data were implicitly shared and shared data had to be explicitly protected. Compared to Perl interpreter threads, 5005threads are more unstable. Regular expression capture variables, such as `$1`, are not thread safe and are easily corrupted by competing threads.

2.1.10 NESL

NESL [22] combines aspects of strict data-parallelism languages and control-parallelism languages. NESL uses flatten-nested-parallelism to map control-parallel programs to data-parallel architectures. NESL is a small-core, side-effect-free language, where every parallel expression is compiled into a parallel form.

2.1.11 OpenMP

OpenMP [33] is a collection of compiler directives that enable user-directed, shared-memory parallelism in C, C++, and Fortran. The OpenMP `parallel` directive annotates a region of code as a parallel region that will be executed by multiple threads. OpenMP does not provide any

safety properties. Programmers are responsible for using the API correctly and creating conforming programs.

2.1.12 KaffeOS

KaffeOS [2, 3] is a Java runtime system, which enables process like isolation of multiple applications with in a single Java virtual machine. Each process executes as if it were in its own virtual machine. Each KaffeOS process has its own heap. A limited form of shared memory communication is allowed via shared heaps. Objects in a shared heap are forbidden from having references back into user process heaps. KaffeOS implements a user kernel boundary and allows limits on individual processes resource consumption. KaffeOS outperforms commercial JVMs in the presence of denial-of-service or misbehaving code but performs slower when programs are well behaved.

2.2 Hybrid Parallelism Languages

2.2.1 Partitioned Global Address Space (PGAS)

Partitioned Global Address Space (PGAS) [51] languages use the convenient shared-memory model as an abstraction for message passing. PGAS languages have local pointers that point to a process' local memory and global pointers which can point to remote or local variables. Communication in PGAS languages is one sided; processes put and get values directly into remote process' memory, without involving the remote application.

Two of the most common PGAS languages are Unified Parallel C (UPC) [49] and Titanium [25] (a scientific computing dialect of Java). Both of these PGAS languages are implemented as source-to-source transformations which compile down to C code.

2.2.2 X10

X10 [10] is a partitioned global address space (PGAS) language whose sequential language is largely taken from Java. Although our use of the term "place" is inspired by X10, places are more static in X10, in that the number of places within an X10 program is fixed at startup. Like Racket places, objects that exist at an X10 place are normally manipulated only by tasks within the place. X10 includes an `at` construct that allows access to an object in one place from another place, so `at` is effectively the communication construct for places in X10. Racket's message-passing communication is more primitive, but also more directly exposes the cost of cross-place communication. Features similar to X10's cross-place references and `at` could be implemented on top of Racket's message-passing layer.

2.2.3 High-Performance Fortran

High-Performance Fortran [38] is a programming language designed for data parallel programming. HPF allows the programmer to specify data alignment and data distribution. HPF also contains a rich set of parallel array assignment and manipulation statements.

2.3 Languages with Distributed Parallelism

2.3.1 Erlang

Erlang's [40] distributed capabilities are built upon its process concurrency model. Remote Erlang nodes are identified by `name@host` identifiers. New Erlang processes can be started using the `slave:start` procedure or at the command line. Erlang uses a feature called links to implement fault notification. Two processes establish a link between themselves. Links are bidirectional; if either process fails the other process dies also. Erlang also provides monitors which are unidirectional notifications of a process exiting. Distributed Places and Erlang share a lot of similar features. While Erlang's distributed processes are an extension of its process concurrency model, Distributed Places are an extension of Racket's places parallelism strategy. Erlang provides a distributed message passing capability that integrates transparently with its interprocess message passing capability. The Disco project implements map reduce on top of a Erlang core. User level Disco programs, however, are written in Python, not Erlang. In contrast, the implementation and user code of distributed places' map reduce are both expressed as Racket code. Erlang has a good foundation for building higher-level distributed computing frameworks, but instead Erlang programmers seem to build customized distributed solutions for each application.

2.3.2 MapReduce

MapReduce [13] is a specialized functional programming model, where tasks are automatically parallelized and distributed across a large cluster of commodity machines. MapReduce programmers supply a set of input files, a map function, and a reduce function. The map function transforms input key/value pairs into a set of intermediate key/value pairs. The reduce function merges all intermediate values with the same key. The framework does all the rest of the work. Google's MapReduce implementation handles partitioning of the input data, scheduling tasks across distributed computers, restarting tasks due to node failure, and transporting intermediate results between compute nodes. The MapReduce model can be applied to problems such as word occurance counting, distributed grep, inverted index creation, and distributed sort.

2.3.3 Termite

Termite [21] is a distributed concurrent scheme built on top of Gambit-C Scheme. Direct mutation of variables and data structures is forbidden in Termite. Instead, mutation is simulated using messages and suspended, lightweight processes. Lookup in Termite's global environment is a node relative operation and resolves to the value bound to the global variable on the current node. Termite supports process migration via serializable closures and continuations. Termite follows Erlang's style of failing hard and fast. Where Erlang has bidirectional links, Termite has directional links that communicate process failure from one process to another. Failure detection only occurs in one direction from the process being monitored to the monitoring process. Termite also has supervisors which, like supervisors in Erlang, restart child processes which have failed. Distributed Places could benefit from Termites superior serialization support, where nearly all Termite VM objects are serializable.

2.3.4 Akka

Akka [48] is a concurrency and distributed processing framework for Scala and Java. Like Erlang, Akka is patterned after the Actor model. Akka supports Erlang like supervisors and monitors for failure and exit detection. Like Erlang, Akka leaves the creation of higher-level distributed frameworks to custom application developers.

2.3.5 Kali

Kali [9] is a distributed version of Scheme 48 that efficiently communicates procedures and continuations from one compute node to another. Kali's implementation lazily faults continuation frames across the network as they are needed. Kali's proxies are really just address space relative variables. Proxies are identified by a globally unique id. Sending a proxy involves sending only its globally unique id. Retrieving a proxies value returns the value for the current address space. Kali allow for retrieval of the proxy's source node and spawning of new computations at the proxy's source.

2.3.6 Distributed Functional Programming in Scheme (DFPS)

Distributed Functional Programming in Scheme (DFPS) [43] uses futures semantics to build a distributed programming platform. DFPS employs the Web Server collection's `serial-lambda` form to serialize closures between machines. Unlike Racket futures, DFPS' `touch` form blocks until remote execution of the future completes. DFPS has a distributed variable construct called a `dbox`. For consistency, a dbox should only be written to once or a reduction function for writes to

the dbox should be provided. Once a dbox has be set, the DFPS implementation propagates the dbox value other nodes that reference the dbox,

2.3.7 Cloud Haskell

Cloud Haskell [15, 16] is a distributed programming platform built in Haskell. Cloud Haskell has two layers of abstraction. The lowest layer is the process layer, which is a message-passing distributed programming API. Next comes the tasks layer, which provides a framework for failure recovery and data locality. Communication of serialized closures requires explicit specification from the user of what parts of environment will be serialized and sent with the code object.

On top of its message-passing process layer, Cloud Haskell implements typed channels that allow only messages of a specific type to be sent down the channel. A Cloud Haskell channel has a SendPort and a ReceivePort. ReceivePorts are not serializable and cannot be shared, which simplifies routing. SendPorts, however, are serializable and can be sent to multiple processes, allowing many to one style communication.

2.3.8 High-level Distributed-Memory Parallel Haskell (HdpH)

High-level Distributed-Memory Parallel Haskell (HdpH) [27] builds upon Cloud Haskell's work by adding support for polymorphic closures and lazy work stealing. HdpH does not require a special language kernel or any modifications to the vanilla GHC runtime. It simply uses GHC's Concurrent Haskell as a systems language for building a distributed memory Haskell.

2.3.9 Dryad

Dryad [26] is an infrastructure for writing coarse-grain data-parallel distributed programs on the Microsoft platform. Distributed programs are structured as a directed graph. Sequential programs are the graph vertices and one-way channels are the graph edges. Unlike Distributed Places, Dryad is not a programming language. Instead, it provides a execution engine for running sequential programs on partitioned data at computational vertices. Although Dryad is not a parallel database, the relational algebra can be mapped on top of a Dryad distributed compute graph. Unlike distributed places, which is language centric, Dryad is infrastructure piece, which does not extend the expressiveness of any particular programming language.

2.3.10 Jade

Jade [39] is a implicitly parallel language. Implemented as a extension to C, Jade is intended to exploit task-level concurrency. Like OpenMP, Jade consists of annotations that programmers add

to their sequential code. Jade uses data access and task granularity annotations to automatically extract concurrency and parallelize the program. A Jade front end then compiles the annotated code and outputs C. Programs parallelized with Jade continue to execute deterministically after parallelization. Jade's data model can interact badly with the programs that write to disjoint portions of a single aggregate data structure. In contrast, Distributed Places is an explicitly parallel language where the programmer must explicitly spawn tasks and explicitly handle communication between tasks.

2.3.11 Dreme

Dreme [20] is a distributed Scheme. All first-class language objects in Dreme are mobile in the network. Dreme describes the communication network between nodes using lexical scope and first class closures. Dreme has a network-wide distributed memory and a distributed garbage collector. By default, Dreme sends objects by reference across the network, which can lead to large quantities of hidden remote operations. In contrast, distributed places copies all objects sent across the network and leaves the programmer responsible for communication invocations and their associated costs.

2.4 Transactional Memory

TransactionalMemory [23, 36] allows a set of memory loads and stores to execute atomically. Transactional memory is an optimistic, lock-free form of atomicity that logs loads and stores during the execution of a transaction body. At the end of a transaction, if the load and store addresses have not been modified by another thread, the stores are committed to memory. When another thread modifies memory addresses used within the transaction, the transaction aborts and retires execution at the beginning of the transaction body. Transactional memory provides the benefits of nonblocking behavior and wait-freedom while allowing programmers to use a critical section style for writing programs.

CHAPTER 3

PLACES

3.1 Introduction

The increasing availability of multicore processors on commodity hardware—from cell phones to servers—puts increasing pressure on the design of dynamic languages to support multiprocessing. Support for multiprocessing often mimics the underlying hardware: multiple threads of execution within a shared address space. Unfortunately, the problem of developing correct parallel programs with threads of execution in a single address space are well known, nontrivial, and afflict both programmers using a language and the implementors of the language. Programmers and language implementors alike need better alternatives.

A message-passing architecture, with threads of execution in separate address spaces, is widely recognized as a more scalable design and easier to reason about than shared memory. Besides avoiding the interference problems created by shared memory, the message-passing model encourages programmers to consider the data-placement and communication needs of a program to enable sustained scalability. The design and success of languages like Erlang demonstrate the viability of this model for parallel programming.

Racket's new *place*[1] construct supports message-passing parallelism layered on top of a language that (unlike Erlang) was not originally designed for parallelism. Racket's existing threads and synchronization support for *concurrency* are kept separate from new places support for *parallelism*, except to the degree that message receipt interacts with other concurrent activities within a single place. Message-passing parallelism is not novel in Racket, but our design and experience report for layering places on top of an existing language should be useful to other designers and implementors.

The conventional approach to adding this style of parallelism to a language implementation that has a large, sequential runtime system is to exploit the UNIX `fork()` primitive, much in the way Python's multiprocessing library works. This approach, however, limits the communication between cooperating tasks to byte streams, making abstraction more difficult and communication

[1]The choice of the name "place" is inspired by X10's construct. [10]

less efficient than necessary. The decision was made to implement places directly in the runtime system, instead of relying on the operating system. This approach allows the runtime system to maintain more control and also fits our ongoing effort to explore the boundary between the operating system and the programming language [17, 18, 50].

The Racket runtime system begins with a single, initial place. A program can create additional places, send messages to places over channels—including channels as messages, so that any two places can communicate directly. Messages sent between places are normally immutable, preventing the data races that plague shared-memory designs. To allow lower-level communication when appropriate, however, places can share certain mutable data structures, including byte strings, fixnum arrays, and floating-point arrays, all of which contain only atomic values.

As part of Racket's broader approach to parallelism, places fully support the previously reported construct for parallelism, *futures* [45]. In particular, each place can spawn and manage its own set of future-executing threads. Places and futures are complementary; places support coarse-grained parallelism without restrictions on the parallel computations, while futures support fine-grained parallelism for sufficiently constrained computations (e.g., no I/O).

The rest of the chapter proceeds as follows. Section 3.2 explains in more detail the design rationale for places. Section 3.3 briefly outlines the places API. Section 3.4 demonstrates how message passing, shared memory, and higher-level parallelism constructs can be built on top of place primitives. Section 3.5 explains the implementation of places within the Racket virtual machine. Section 3.7 evaluates the performance and scaling of places using the NAS Parallel Benchmarks.

3.2 Design Overview

Each *place* is essentially a separate instance of the Racket virtual machine. All code modules are loaded separately in each place, data are (almost always) allocated in a specific place, and garbage collection proceeds (almost always) independently in each place.

Places communicate through *place channels*, which are endpoints for communication channels that are shared among processes in much the way that Unix processes use file descriptors for endpoints of shared pipes. Unlike file descriptors, a place channel supports structured data across the channel, including booleans, numbers, characters, symbols, byte strings, Unicode strings, filesystem paths, pairs, lists, vectors, and "prefab" structures (i.e., structures that are transparent and whose types are universally named). Roughly speaking, only immutable data can be sent across a place channel, which allows the implementation to either copy or share the data representation among places as it sees fit. Place channels themselves can be sent in messages across place channels, so

that communication is not limited to the creator of a place and its children places; by sending place channels as messages, a program can construct custom message topologies.

In addition to immutable values and place channels, special mutable byte strings, fixnum vectors, and floating-point vectors can be sent across place channels. For such values, the runtime system is constrained to share the underlying value among places, rather than copy the value as it is sent across a channel. Mutation of the value by one place is visible to other places. By confining shared mutable values to vectors of atomic data, race conditions inherent in sharing cannot create safety problems for the runtime system or complicate garbage collection by allowing arbitrary references from one address space to another. At the same time, shared vectors of atomic data directly support many traditional parallel algorithms, such as a parallel prefix sum on a vector of numbers. Other mutable values could be allowed in place messages with the semantics that they are always copied, but such copying might be confusing, and explicit marshaling seems better to alert a programmer that copying is unavoidable (as opposed to any copying that the runtime system might choose as the best strategy for a given message).

The prohibition against sharing arbitrary mutable values implies that thunks or other procedures cannot be sent from one place to another, since they may close over mutable variables or values. Consequently, when a place is created, its starting code is not specified by a thunk (as is the case for threads) but by a module path plus an exported "main" function. This specification of a starting point is essentially the same as the starting point in Racket itself, except that the "main" function receives a place channel to initiate communication between the new place and its creator. The `place` form simplifies place creation where a procedure would be convenient, but it works by lifting the body of the `place` form to an enclosing module scope at compile time.

Additional place channels can be created and sent to places, allowing the creation of specific constructed capabilities. One common pattern is to have a master place spawn worker places and collect all of the initial place-channels into a list. This list of place channels can then be sent to all the places, which permits all-to-all communication. Place channels are asynchronous, so that the sender of a message need not synchronize with a recipient. Place channels are also two-way as a convenience; otherwise, since a typical communication patterns involve messages in both directions, a program would have to construct two place channels. Finally, place channels are *events* in the sense of Concurrent ML [37, 17]. Place channels can be combined with other events to build up complex synchronization patterns, such as fair choice among multiple place channels.

Our current initial implementation of places shares little read-only data among places. Longer term, it would be nice to automatically share read-only code modules and JIT-generated code across

places in much the same way that operating systems share libraries among separate applications. In general, places are designed to allow such sharing optimizations in the language runtime system as much as possible.

3.3 Places API

The Racket API for places[2] supports place creation, channel messages, shared mutable vectors, and a few administrative functions.

```
(dynamic-place module-path start-proc) → place?
  module-path : module-path?
  start-proc : symbol?
```

creates a place to run the procedure that is identified by *module-path* and *start-proc*.[3] The result is a place descriptor value that represents the new parallel task; the place descriptor is returned immediately. The place descriptor is also a place channel to initiate communication between the new place and the creating place.

The module indicated by *module-path* must export a function with the name *start-proc*. The exported function must accept a single argument, which is a place channel that corresponds to the other end of communication for the place channel that is returned by `dynamic-place`. For example,

```
(dynamic-place "fib.rkt" 'go)
```

starts the module `"fib.rkt"` in a new place, calling the function `go` that is exported by the module.

```
(place id body ...+)
```

The `place` derived form creates a place that evaluates *body* expressions with *id* bound to a place channel. The *body*s close only over *id* plus the top-level bindings of the enclosing module, because the *body*s are lifted to a function that is exported by the module. The result of `place` is a place descriptor, like the result of `dynamic-place`.

For example, given the definitions

---

[2]This section describes the API of places for the 5.1.2 release version of Racket at `http://racket-lang.org/download/`.

[3]The `dynamic-` prefix on the function name reflects the similarity of this function to Racket's `dynamic-require` function.

```
(define (fib n) ....)

(define (start-fib-30)
  (place ch (fib 30)))
```

then calling `start-fib-30` creates a place to run a new instantiation of the enclosing module, and the `fib` function (which need not be exported) is called in the new place.

(`place-channel-put` *ch v*) → `void?`
  ch : `place-channel?`
  v : `place-message-allowed?`
(`place-channel-get` *ch*) → `place-message-allowed?`
  ch : `place-channel?`

The `place-channel-put` function asynchronously sends a message *v* on channel *ch* and returns immediately. The `place-channel-get` function waits until a message is available from the place channel *ch*. See also `sync` below.

As an example, the following `start-fib` function takes a number *n*, starts `(fib` *n*`)` in a new place, and returns a place descriptor to be used as a place channel for receiving the result:

```
(define (fib n) ....)

(define (start-fib n)
  (define p
    (place ch
      (define n (place-channel-get ch))
      (place-channel-put ch (fib n))))
  (place-channel-put p n)
  p)
```

The `start-fib` function could be used to start two computations in parallel and then get both results:

```
(define p1 (start-fib n1))
(define p2 (start-fib n2))
(values (place-channel-get p1)
        (place-channel-get p2))
```

(`place-channel-put/get` *ch v*)
 → `place-message-allowed?`
  ch : `place-channel?`
  v : `place-message-allowed?`

A convenience function to combine a `place-channel-put` with an immediate `place-channel-get`.

```
(place-channel) → place-channel? place-channel?
```

returns two place channels that are cross-linked through an underlying data channel. Data sent through the first place channel are received through the second place channel and vice versa.

For example, if `buyer` and `seller` places are given channel endpoints, they can communicate directly using the new channel and report only final results through their original channels:

```
(define b (dynamic-place "trade.rkt" 'buyer))
(define s (dynamic-place "trade.rkt" 'seller))

(define-values (b2s s2b) (place-channel))
(place-channel-put b b2s)
(place-channel-put s s2b)
; ... buyer and seller negotiate on their own ...

(values (place-channel-get b)
        (place-channel-get s))
```

```
(sync evt ...+) → any?
  evt : evt?
```

blocks until at least one of the argument *evt*s is ready, and returns the value of the ready *evt*. A place channel as an event becomes ready when a message is available for the channel, and the corresponding value produced by `sync` is the channel message. Thus, `(sync ch1 ch2)` receives a message from `ch1` or `ch2`—whichever has a message first.

Racket includes other synchronization constructs, such as the `sync/timeout` function to poll an event. Our examples in this paper need only `sync`.

```
(handle-evt evt handle) → handle-evt?
  evt : (and/c evt? (not/c handle-evt?))
  handle : (any/c . -> . any)
```

creates an event that is in a ready when *evt* is ready, but whose result is determined by applying *handle* to the result of *evt*.

```
(place-wait p) → void?
  p : place?
```

blocks until *p* terminates.

```
(make-shared-fxvector size [x]) → fxvector?
  size : exact-nonnegative-integer?
  x : fixnum? = 0
(make-shared-flvector size [x]) → flvector?
  size : exact-nonnegative-integer?
  x : flonum? = 0.0
```

creates a mutable, uniform vector of fixnums or floating-point numbers that can be shared across places. That is, the vector is allowed as a message on a place channel, and mutations of the vector by the sending or receiving place are visible to the other place. The concurrency model for shared data is determined by the underlying processor (e.g., TSO [44] for x86 processors). Places can use message passing or functions like `place-wait` to synchronize access to a shared vector.

For example,

```
(define (zero! vec)
  (define p
    (place ch
      (define vec (place-channel-get ch))
      (for ([i (fxvector-length vec)])
        (fxvector-set! vec i 0))))
  (place-channel-put p vec)
  (place-wait p))
```

fills a mutable fixnum vector with zeros using a separate place. Waiting until the place is finished ensures that the vector is initialized when `zero!` returns.

`(processor-count)` → `exact-positive-integer?`

returns the number of parallel computation units (e.g., processors or cores) that are available on the current machine.

3.4 Design Evaluation

The evaluation of the design of places proceeds in two main ways. First, places was used for Racket's parallel-build infrastructure, where the implementation uses Erlang-style message handling. Second, the NAS Parallel Benchmark suite were ported to Racket using MPI-like parallelism constructs that are built on top of places. In addition to the main experiments, a Mandelbrot example is presented that demonstrates how atomic-value vectors can be shared among places. Together, these examples demonstrate the versatility of places for implementing different patterns of parallelism.

3.4.1 Parallel Build

The full Racket source repository includes 700k lines of Racket code plus almost 200k of documentation source (which is also code) that is recompiled with every commit to the repository. A full build takes nearly an hour on a uniprocessor, but the build parallelizes well with places, speeding up by 3.2x on 4 cores.

The build is organized as a controller in the main place that spawns workers in their own places. One worker is created for each available processor. The controller keeps track of the files that need to be compiled, while each worker requests a file to compile, applies the `compile` function to the file, and repeats until no more files are available from the controller.

Concretely, the workers are created by `place` in a `for/list` comprehension that is indexed by an integer from `0` to `(processor-count)`:

```
(define ps ;list of place descriptors
  (for/list ([i (processor-count)])
    (place ch
      (let worker ()
        (match (place-channel-put/get ch 'get-job)
          ['done (void)]
          [job
           (compile job)
           (define msg (list 'job-finished job))
           (place-channel-put ch msg)
           (worker)])))))
```

Each worker runs a `worker` loop that sends a `'get-job` message to the controller via `ch` and then waits for a response. If the response to the `'get-job` request is the symbol `'done`, the controller has no more jobs; the place quits running by returning `(void)` instead of looping. If the controller responds with a job, the worker compiles the job, sends a completion message back to the controller, and loops back to ask for another job.

After spawning workers, the controller waits in a loop for messages to arrive from the workers. Any worker might send a message, and the controller should respond immediately to the first such message. In Concurrent ML style, the loop is implemented by applying `sync` to a list of events, each of which wraps a place channel with a handler function that answers the message and recurs to the message loop. When no jobs are available to answer a worker's request, the worker is removed from the list of active place channels, and the message loop ends when the list is empty.

The message-handling part of the controller matches a given message `m`, handles it, and recurs via `message-loop` (lines 21-33 in Figure 3.1). Specifically, when the controller receives a `'get-job` message, it extracts a job from the job queue. If the job queue has no remaining jobs so that `(get-job job-queue)` returns `#false`, the `'done` message is sent to the worker; otherwise, the job from the queue is sent back to the worker. When the controller instead receives a `(list 'job-finished job)` message, it notifies the job queue of completion and resumes waiting for messages.

Figure 3.1 contains the complete parallel-build example. Racket's actual parallel-build imple-

```
#lang racket
(require "job-queue.rkt")

(define (main)
  (define ps ;list of place descriptors
    (for/list ([i (processor-count)])
      (place ch
        (let worker ()
          (place-channel-put ch 'get-job)
          (match (place-channel-get ch)
            ['done (void)]
            [job
             (compile job)
             (define msg (list 'job-finished job))
             (place-channel-put ch msg)
             (worker)])))))

  (define job-queue (build-job-queue))

  (define (make-message-handler p ps)
    (define (message-handler m)
      (match m
        ['get-job
         (match (get-job job-queue)
           [#false
            (place-channel-put p 'done)
            (message-loop (remove p ps))]
           [job
            (place-channel-put p job)
            (message-loop ps)])]
        [(list 'job-finished job)
         (job-finished job-queue job)
         (message-loop ps)]))
    (handle-evt p message-handler))

  (define (message-loop ps)
    (define (make-event p)
      (make-message-handler p ps))
    (unless (null? ps)
      (apply sync (map make-event ps))))

  (message-loop ps))
```

Figure 3.1: Parallel Build

mentation is more complicated to handle error conditions and the fact that compilation of one module may trigger compilation of another module; the controller resolves conflicts for modules that would otherwise be compiled by multiple workers.

3.4.2 Higher-level Constructs

Repeatedly creating worker modules, spawning places, sending initial parameters, and collecting results quickly becomes tiresome for a Racket programmer. Racket's powerful macro system, however, permits the introduction of new language forms to abstract such code patterns. The Racket version of the NAS parallel benchmarks are built using higher-level constructs: `fork-join`, `CGfor` and `CGpipeline`.

3.4.2.1 CGfor

The `CGfor` form looks like the standard Racket `for` form, except for an extra *communicator group* expression. The communicator group records a configuration in three parts: the integer identity of the current place, the total number of places in the communicator group, and a vector of place channels for communicating with the other places. The `CGfor` form consults a given communicator group to partition the loop's iteration space based on the number of places in the group, and it executes the loop body only for indices mapped to the current place's identity. For example, if a communication group `cg` specifies 3 places, then `(CGfor cg ([x (in-range 900)]) ...)` iterates a total of 900 times with the first place computing iterations 1–300, the second place iterating 301–600, and the third place iterating 601–900.

The `fork-join` form creates a communicator group and binds it to a given identifier, such as `cg`. The following example demonstrates a parallel loop using `fork-join` and `CGfor`, which are defined in the `"fork-join.rkt"` library:

```
1 #lang racket
2 (require "fork-join.rkt")
3
4 (define (main n)
5   (fork-join (processor-count) cg ([N n])
6     (CGfor cg ([i (in-range N)])
7       (compute-FFT-x i))
8     (CGBarrier cg)
9     (CGfor cg ([i (in-range N)])
10      (compute-FFT-y i))
11    (CGBarrier cg)
12    (CGfor cg ([i (in-range N)])
13      (compute-FFT-z i))))
```

The `fork-join` form on line 5 creates `(processor-count)` places and records the configuration in a communicator group `cg`. The `([N n])` part binds the size `n` from the original place to `N` in each place, since the new places cannot access bindings from the original place. The `(CGBarrier cg)` expression blocks until all of the places in the communication group `cg` reach the barrier point.

The complete implementation for `fork-join` is shown in Figure 3.2. First `fork-join` spawns places (line 4), sends a message to each place containing the place's identity and other communication-group parameters, and other arguments specified in the `fork-join` use (line 13). It then waits for each place to report its final result, which is collected into a vector of results (line 18).

Each worker place waits for a message from its controller containing its communicator group settings and initial arguments (lines 7-8). The place builds the local communicator group structure (line 9) and evaluates the `fork-join` body with the received arguments (line 10). Finally, the result of the place worker's computation is sent back across a place channel to the place's controller (line 11).

```
1 (define-syntax-rule
2   (fork-join NP cg ([params args] ...) body ...)
3     (define ps
4       (for/list ([i (in-range n)])
5         (place ch
6           (define (do-work cg params ...) body ...)
7           (match (place-channel-get ch)
8             [(list-rest id np ps rargs)
9              (define cg (make-CG id np (cons ch ps)))
10             (define r (apply do-work cg rargs))
11             (place-channel-put ch r)])))))
12
13    (for ([i (in-range NP)] [ch ps])
14      (place-channel-put
15       ch
16       (list i NP ps args ...)))
17
18    (for/vector ([i (in-range NP)] [ch ps])
19      (place-channel-get ch)))
```

Figure 3.2: fork-join

3.4.2.2 CGpipeline

In the same way a `CGfor` form supports simple task parallelism, a `CGpipeline` form supports pipeline parallelism. For example, the LU benchmark uses a parallel pipeline to compute lower and upper triangular matrices. As a simpler (and highly contrived) example, the following code uses pipeline parallelism to compute across the rows of a matrix, where a cell's new value is the squared sum of the cell's old value and the value of the cell to its left. Instead of treating each row as a task, each column is a task that depends on the previous column, but rows can be pipelined through the columns in parallel:

```
1 (define v (flvector 0.0 1.0 2.0 3.0 4.0
2                     0.1 1.1 2.1 3.1 4.1
3                     0.2 1.2 2.2 3.2 4.2
4                     0.3 1.3 2.3 3.3 4.3
5                     0.4 1.4 2.4 3.4 4.4))
6
7 (fork-join 5 cg ()
8   (for ([i (in-range 5)])
9     (CGpipeline cg prev-value 0.0
10      (define idx (+ (* i 5) (CG-id cg)))
11      (define (fl-sqr v) (fl* v v))
12      (fl-sqr (fl+ (fl-vector-ref v idx)
13                   prev-value)))))
```

The pipeline is constructed by wrapping the `CGpipeline` form with a normal `for` loop inside `fork-join`. The `fork-join` form creates five processes, each of which handles five rows in a particular column. The `CGpipeline` form within the `for` loop propagates the value from previous column—in the variable `prev-value`, which is `0.0` for the first column—to compute the current column's value. After a value is produced for a given row, a place can proceed to the next row while its value for the previous row is pipelined to later columns. Like the `CGfor` form, the `CGpipeline` form uses a communicator group to discover a place's identity, the total number of places, and communication channels between places.

Figure 3.3 shows the implementation of `CGpipeline`. All places except place `0` wait for a value from the previous place, while place `0` uses the specified initial value. After place i finishes executing its body, it sends its result to place $i+1$, except for the final place, which simply returns its result. Meanwhile, place i continues to the next row, enabling parallelism through different places working on different rows.

```
(define-syntax-rule
  (CGpipeline cg prev-value init-value body ...)
  (match cg
    [(CG id np pls)
     (define (send-value v)
       (place-channel-put (list-ref pls (add1 id)) v))
     (define prev-value
       (if (= id 0)
           init-value
           (place-channel-get (car pls))))
     (define result (begin body ...))
     (unless (= id (sub1 np)) (send-value result))
     result]))
```

Figure 3.3: CGpipeline

3.4.3 Shared Memory

Certain algorithims benefit from shared-memory communication. Places accommodates a subset of such algorithms through the use of shared vectors. Shared-vector primitives permit a restricted form of shared-memory data structures while preserving the integrity of the language virtual machine. Shared vectors have two integrity-preserving invariants: their sizes are fixed at creation time, and they can only contain atomic values.

In the following Figure 3.4, the `mandelbrot-point` function is a black-box computational kernel. It consumes an (*x, y*) coordinate and returns a Mandelbrot value at that point. The argument `N` specifies the number lines and columns in the output image.

In this implementation, workers communicate `mandelbrot-point` results to the controller through a shared byte vector `b`. Vector `b`'s size is fixed to `(* N N)` bytes, and all `b`'s elements are initialized to 0. The `fork-join` construct spawns the worker places, creates the communicator group `cg`, and sends the line length (`N`) and the shared result vector (`b`) to the workers.

Having received their initial parameters, each place computes its partition of the Mandelbrot image and stores the resulting image fragment into the shared vector (`b`). After all of the worker places finish, the controller prints the shared vector to standard output. The shared-memory implementation speeds up Mandelbrot by 3x on 4 cores.

```
1 #lang racket
2 (require "fork-join.rkt"
3          "mandelbrot-point.rkt")
4
5 (define (main N)
6   (define NP (processor-count))
7   (define b (make-shared-bytes (* N N) 0))
8
9   (fork-join NP cg ([N N] [b b])
10    (CGfor cg ([y (in-range N)])
11      (for ([x (in-range N)])
12        (define mp (mandelbrot-point x y N))
13        (byte-2d-array-set! b x y N mp))))
14
15  (for ([y (in-range N)])
16    (write-bytes/newline b y N)))
```

Figure 3.4: Shared Memory Mandelbrot

3.5 Implementing Places

Prior to support for places, Racket's virtual machine used a single garbage collector (GC) and single OS thread as shown in Figure 3.5. Although Racket has always supported threads, Racket threads support concurrency rather than parallelism; that is, threads in Racket enable organizing a program into concurrent tasks, but threads do not provide a way to exploit multiprocessing hardware to increase a program's performance. Indeed, although threads are preemptive at the Racket level, they are co-routines within the runtime system's implementation.

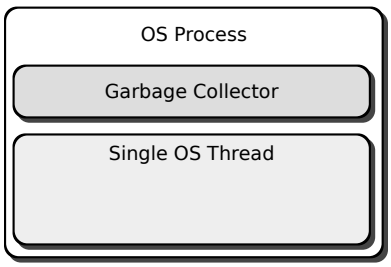

Figure 3.5: Sequential Racket VM

Racket with places uses OS-scheduled threads within the Racket virtual machine. Each place is essentially an instance of the sequential, pre-places virtual machine. To achieve the best parallel performance, places are as independent and loosely coupled as possible, even to the point of separating memory regions among places to maximize locality within a place. Even better, separate address spaces, in Figure 3.6, mean that each place has its own GC that can collect independently from other places.

In Figure 3.7, each place-local GC allocates and manages almost all of the objects that a place uses. An additional master GC is shared across all places to manage a few global shared objects, such as read-only immortal objects, place channels, and shared vectors of atomic values. Object references from places to the shared master heap are permitted, but references are not permitted in the opposite direction.

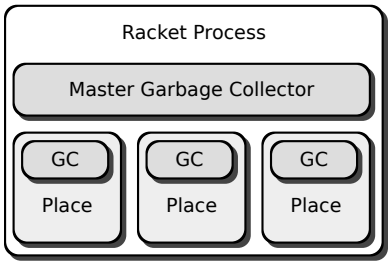

Figure 3.6: Parallel Racket VM

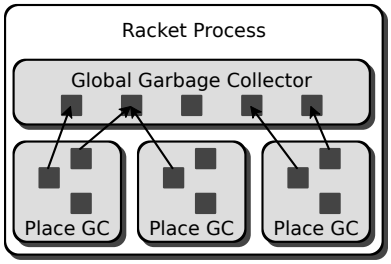

Figure 3.7: GC Object References

Disallowing references from the master space to place-specific spaces maintains isolation between places, and it is the invariant that allows places to garbage collect independently of one another. Only a global collection of the shared master space requires the collective cooperation of all the places, and such collections are rare.

The implementation of places thus consists of several tasks: adding OS schedulable threads to the runtime system, converting global state variables within the runtime system to place-local variables, modifying garbage collection strategies for concurrent-place execution, and implementing channels for communicating between places.

3.5.1 Threads and Global Variables

The Racket runtime system has been continuously developed for the past decade and a half. Like other mature runtime systems, the Racket implementation includes many global variables. The presence of such global variables in the code base was the largest obstacle to introducing OS-scheduled threads into the runtime system.

Using grep and a simple CIL [32] analysis, I conducted an audit of the 719 global variables within the Racket implementation. The audit found 337 variables that fell into the category of read-only singleton objects once they were set (during VM initialization). A few of the variables encountered during the audit, such as `scheme_true`, `scheme_false`, and `scheme_null`, were easy to identify as read-only singleton objects. These were annotated with a `READ_ONLY` tag as documentation and to support further analysis. The auditing of most variables, however, required locating and reviewing all code sites where a particular variable was referenced. About 155 global variables were deemed permissible to share and annotated as `SHARED_OK`. The remaining 227 variables needed to be localized to each place and were tagged as `THREAD_LOCAL_DECL`.

Tool support simplifies the arduous task of annotating and auditing global variables. Tools that simply identify all global variables are remarkably helpful in practice. Finding all the code sites where a global variable is used helps the runtime developer ensure that isolation invariants are preserved in each place that a global variable is referenced.

Testing the global variable audit was relatively easy. The entire Racket test suite was ran in multiple places simultaneously. For almost all global variables that I overlooked or misclassified, parallel execution of the test suite identified the problem.

3.5.2 Thread-Local Variables

To prevent collisions from concurrent access, many global variables were localized as place-specific variables. I considered moving all global variables into a structure that is threaded through

the entire runtime system. Although this restructuring is clean in principle, restructuring the runtime system along those lines would have required extensive modifications to function signatures and code flow. Instead, I decided to use thread-local variables, as supported by the OS, to implement place-local state.

OSes support thread-local variables through library calls, such as `pthread_get_specific()` and `pthread_put_specific()`, and sometimes through compiler-implemented annotations, such as `__threadlocal` or `__declspec(thread)`. Compiler-implemented thread-local variables tend to be much faster, and they work well for Racket on Linux and most other variants of Unix. Although Windows supports compiler-implemented thread-local variables, Windows XP does not support them within DLLs (as used by Racket); Vista and later Windows versions remedy this problem, but Racket 32-bit builds must work on older versions of Windows. Finally, Mac OS X does not currently support compiler-implemented thread-local variables.

Our initial experiments indicated that using library calls for thread-local variables on Windows and Mac OS X would make the runtime system unacceptably slow. Reducing the cost of thread-local variables on those platforms requires two steps.

First, all place-local variables were first collected into a single table. Each place-local variable, such as `toplevels_ht`, has an entry in the table with an underscore suffix:

```
struct Thread_Locals {
  struct Scheme_Hash_Table *toplevels_ht_;
  ....
};

inline struct Thread_Locals *GET_TLV() { ... }

#define toplevels_ht (GET_TLV()->toplevels_ht_)
```

A preprocessor definition for each variable avoids the need to change uses in the rest of the source. Collecting all thread-local variables into a table supports threading a pointer to the table through the most performance-sensitive parts of the runtime system, notably the GC. Along similar lines, JIT-generated code keeps a pointer to the thread-local table in a register or in a local variable.

Second, for uses of thread-local variables outside the GC or JIT-generated code, I implement `GET_TLV()` in a way that is faster than calling `pthread_get_specific()`. In 32-bit Windows, a host executable (i.e., the one that links to the Racket DLL) provides a single thread-local pointer to hold the table of thread-local variables; inline assembly in `GET_TLV()` imitates compiler-supported access to the executable-hosted variable. For Mac OS X, `GET_TLV()` contains an inline-assembly version of `pthread_get_specific()` that accesses the table of thread-local variables.

3.5.3 Garbage Collection

At startup, a Racket process creates an initial GC instance and designates it the master GC. Read-only global variables and shared global tables such as a symbol table, resolved-module path table, and the type table are allocated from the master GC. After the prerequisite shared structures are instantiated, the initial thread disconnects from the master GC, spawns its own GC instance, and becomes the first place. After the bootstrapping phase of the Racket process, the master GC does little besides allocating communication channels and shared atomic-value containers.

Places collect garbage in one of two modes: independently, when collecting only the local heap, or cooperatively as part of a global collection that includes the master GC. Place-local GCs collect their local heap without any synchronization; a place collector traverses the heap and marks objects it allocated as live, and all other encountered objects, including objects allocated by the master GC, are irrelevant and ignored.

When the master GC needs to perform a collection, all places must pause and cooperate with the master GC. Fortunately, most allocation from the master GC occurs during the initialization of a program. Thus, the master GC normally reaches a steady state at the beginning of some parallel program, allowing places to run in parallel without interruption in common situations.

To initiate a global collection, the master GC sends a signal to all places asking them to pause mutation and cooperatively collect. Each place then performs a local collection in parallel with one another. During cooperative collection, a place GC marks as live not only traversed objects it allocated but also objects that were allocated by the master GC; races to set mark bits on master-GC objects are harmless. Master-GC objects that are referenced only by place-local storage are thus correctly preserved.

After all place-specific collections have finished, each place waits until the master GC marks and collects. Although place-specific collection can move objects to avoid fragmentation, the master GC never moves objects as it collects; master-GC allocation is rare and coarse-grained enough that compaction is not needed. Each place can therefore resume its normal work as soon as the master-GC collection is complete.

3.5.4 Place Channels

To maintain the invariant that allows the place-specific GCs to work independently, sending a message over a place channel copies data from the originating place to the destination place.

Place channels implement efficient, one-copy message passing by orphaning memory pages from the source place and adopting those memory pages into the destination place. A place channel begins this process by asking its local allocator for a new orphan allocator. The orphan allocator

groups all its allocations onto a new set of orphaned memory pages. Orphaned pages are memory blocks that are not owned by any GC. The place channel then proceeds to copy the entire message using the orphan allocator. After the copy is completed, the new orphaned message only contains references to objects within itself and shared objects owned by the master GC. The originating place sends this new message and its associated orphaned memory pages to the destination place.

A place channel, receiving a message, adopts the message's orphaned memory pages into its own nursery generation and returns the received message to the user program. Message contents that survive the nursery generation will relocate to memory more localized to the receiving place as the objects are promoted from the nursery to the mature object generation. This orphan-adoption process allows for single copy asynchronous message passing without needing to coordinate during message allocation.

Messages less than 1024 bytes in length are handled in a slightly different manner. These short messages are allocated onto an orphan page and sent to the destination place exactly as described above. At the short message's destination, instead of adopting the messages orphaned pages, the destination place copies the message from the orphan page into its local allocator. By immediately copying short messages into the destination place allocator, the orphaned page can be returned to the system immediately for use by subsequent place-channel messages.

The graphs in Figure 3.8 summarize the performance of place-channel communication. The first graph compares `memcpy()` in C, place channels in Racket, and pipes in Racket on a byte-string message. The results, which are plotted on a log scale, show that place channels can be much slower than raw `memcpy()` for small messages, where the cost of memory-page management limits place-channel throughput. Messages closer to a page size produce similar throughput with all techniques. The second graph shows place-channel, pipe, and socket performance when the message is a list, where Racket's `write` and `read` are used to serialize lists for pipes and sockets. The graph shows that place-channel communication remains similar to pipe and socket communication for structured data. Together, the results show that our communication strategy does not make communication particularly cheap, but it is competitive with services that have been optimized by OS implementors.

3.5.5 OS Page-Table Locks

Compilation of Racket's standard library was one of our early tests of performance with places. After eliminating all apparent synchronization points as possible points of contention, I found that using separate processes for the build scaled better than using places within a single Racket process. On closer inspection of the system calls being made in each case, I saw that the build used many

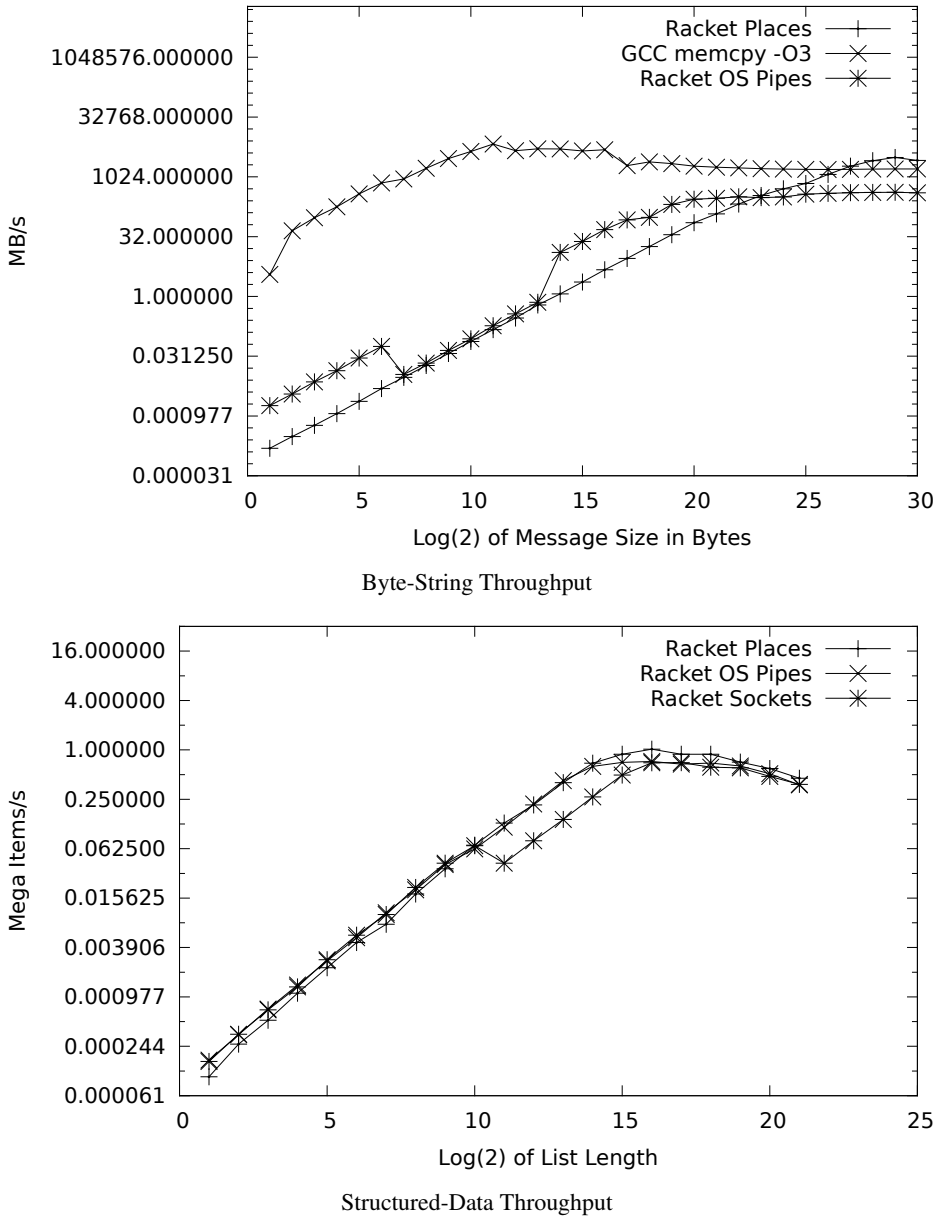

Figure 3.8: Place-Channel Performance

`mprotect()` calls that took a long time to complete.

The Racket generational garbage collector uses OS-implemented memory protection to implement write barriers. Each garbage collection uses `mprotect()` to clear and set read-only permissions on memory pages. After consulting the Linux source code, I realized that `mprotect()` acquires a lock on the process's page table. When two or more places garbage collect at the same

time, contention for the process's page table lock greatly increased the time for `mprotect()` calls to complete. To avoid this problem, I implemented an extra layer for the Racket allocator to produce larger blocks of contiguous mature objects; issuing a single `mprotect()` call for the contiguous block reduces the overall number of `mprotect()` calls by an order of magnitude, which eliminates the bottleneck.

The more general lesson is that OSes are designed to support separate processes with minimal interference, but some corners of an OS rely on relatively heavy locks within a single process. Fortunately, I do not encounter these corners often—for example, concurrent filesystem access seems to perform as well with places as with separate processes—but the possibility is an extra concern for the implementation.

3.5.6 Overlooked Cases and Mistakes

Retrofitting a language virtual machine and garbage collector for parallelism is a huge effort that is bound to result in bugs and overlooked cases. The foreign function interface (FFI) is one example. The FFI contains a `"opened_libs"` hash that holds opened libraries handles. The FFI for Racket lives outside the main VM source directory, `"src/racket/src"`. In auditing the VM source code, I overlooked the FFI and failed to annotate `"opened_libs"` as a place local variable. As a result, multiple places contended over use of `"opened_libs"` until it was marked to be place specific.

There were multiple occasions when I forgot to write garbage collector mark routines for new VM structures or members. In several cases, structures which should have been created as place specific were overlooked because they were lazily created on demand. In another case, when using the Racket event system, a place message could be leaked if an exception was thrown. The in-transit place message needed to be freed before the exception could be allowed to propagate.

Pairs in the Racket VM are a very common and highly optimized data structure. Pairs cache whether they are a member of a list or not by setting bits in their object header . When copying pairs across a place channel, I copied the obvious first and second members of the pair but forgot to copy the list optimization flag bits that were stored in the object header.

The complexity and power of language virtual machines is impressive yet demands respect. Bugs humble all programmers, but garbage collection and virtual machine bugs can have an extra measure of nastiness.

3.5.7 Overall: Harder than it Sounds, Easier than Locks

The conversion of Racket to support places took approximately two graduate-student years, which is at least four times longer than I originally expected. At the same time, the implementation of places has proven more reliable than I expected; when I eventually flipped the default configuration of Racket from no-places (and a parallel library build based on OS processes) to places (and using them for building libraries), our automatic test builds continued exactly as before—with the same success rate and performance. Further deployments uncovered memory leaks, but those were quickly corrected.

Our experience with places contrasts sharply with our previous experience introducing concurrency into the runtime system, where months of additional testing and use were required to uncover many race conditions that escaped detection by the test suite. I attribute this difference primarily to the small amount of sharing among places, and therefore the small number of locking locations and potential races in the code.

While implementing places, I made many mistakes where data from one place were incorrectly shared with another place, either due to incorrect conversion of global variables in the runtime system or an incorrect implementation of message passing. Crashes from such bugs were highly reproducible, however, because a bad reference in a place tends to stick around for a long time, so it is detected by an eventual garbage collection. Bugs due to incorrect synchronization, in contrast, hide easily because they depend on relatively unlikely coincidences of timing that are exacerbated by weak memory models.

In adding places to Racket, I did not find as easy a path to parallelism as I had hoped. I did, however, find a preferable alternative to shared memory and locks.

3.6 Places Complete API

Places enable the development of parallel programs that take advantage of machines with multiple processors, cores, or hardware threads. A *place* is a parallel task that is effectively a separate instance of the Racket virtual machine. Places communicate through *place channels*, which are endpoints for a two-way buffered communication.

To a first approximation, place channels support only immutable, transparent values as messages. In addition, place channels themselves can be sent across channels to establish new (possibly more direct) lines of communication in addition to any existing lines. Finally, mutable values produced by `shared-flvector`, `make-shared-flvector`, `shared-fxvector`, `make-shared-fxvector`, `shared-bytes`, and `make-shared-bytes` can be sent across place channels; muta-

tion of such values is visible to all places that share the value, because they are allowed in a *shared memory space*. See `place-message-allowed?`.

A place channel can be used as a synchronizable event to receive a value through the channel. A place can also receive messages with `place-channel-get`, and messages can be sent with `place-channel-put`. Two place channels are `equal?` if they are endpoints for the same underlying channels while both or neither is a place descriptor. Place channels can be `equal?` without being `eq?` after being sent messages through a place channel.

Constraints on messages across a place channel—and therefore on the kinds of data that places share—enable greater parallelism than `future`, even including separate garbage collection of separate places. At the same time, the setup and communication costs for places can be higher than for futures. For example, the following expression launches two places, echoes a message to each, and then waits for the places to terminate:

```
(let ([pls (for/list ([i (in-range 2)])
             (dynamic-place "place-worker.rkt" 'place-main))])
  (for ([i (in-range 2)]
        [p pls])
    (place-channel-put p i)
    (printf "~a\n" (place-channel-get p)))
  (map place-wait pls))
```

The `"place-worker.rkt"` module must export the `place-main` function that each place executes, where `place-main` must accept a single place channel argument:

```
#lang racket
(provide place-main)

(define (place-main pch)
  (place-channel-put pch (format "Hello from place ~a"
                                 (place-channel-get pch))))
```

`(place-enabled?) → boolean?`

returns `#t` if Racket is configured so that `dynamic-place` and `place` create places that can run in parallel, `#f` if `dynamic-place` and `place` are simulated using `thread`.

`(place? v) → boolean?`
`  v : any/c`

returns `#t` if *v* is a *place descriptor* value, `#f` otherwise. Every place descriptor is also a place channel.

```
(place-channel? v) → boolean?
  v : any/c
```

returns `#t` if *v* is place channel, `#f` otherwise.

```
(dynamic-place module-path start-name) → place?
  module-path : (or/c module-path? path?)
  start-name : symbol?
```

creates a place to run the procedure that is identified by *module-path* and *start-name*. The result is a place descriptor value that represents the new parallel task; the place descriptor is returned immediately. The place descriptor value is also a place channel that permits communication with the place.

The module indicated by *module-path* must export a function with the name `start-proc`. The function must accept a single argument, which is a place channel that corresponds to the other end of communication for the place descriptor returned by `place`.

When the place is created, the initial exit handler terminates the place, using the argument to the exit handler as the place's *completion value*. Use `(exit v)` to immediately terminate a place with the completion value *v*. Since a completion value is limited to an exact integer between `0` and `255`, any other value for `v` is converted to `0`. If the function indicated by *module-path* and `start-proc` returns, then the place terminates with the completion value `0`.

In the created place, the `current-input-port` parameter is set to an empty input port, while the values of the `current-output-port` and `current-error-port` parameters are connected to the current ports in the creating place. If the output ports are file-stream ports, then the connected ports in the places share the underlying stream, otherwise a thread in the creating place pumps bytes to the current ports in the creating place.

The *module-path* argument must not be a module path of the form `(quote sym)` unless the module is predefined (see `module-predefined?`).

```
(dynamic-place* module-path
                start-name
               [#:in in
                #:out out
                #:err err]) → place?
                              (or/c output-port? #f)
                              (or/c input-port? #f)
                              (or/c input-port? #f)
  module-path : (or/c module-path? path?)
  start-name : symbol?
```

```
in : (or/c input-port? #f) = #f
out : (or/c output-port? #f) = (current-output-port)
err : (or/c output-port? #f) = (current-error-port)
```

like `dynamic-place`, but accepts specific ports to the new place's ports, and returns a created port when `#f` is supplied for a port. The *`in`*, *`out`*, and *`err`* ports are connected to the `current-input-port`, `current-output-port`, and `current-error-port` ports, respectively, for the place. Any of the ports can be `#f`, in which case a file-stream port (for an operating-system pipe) is created and returned by `dynamic-place*`. The *`err`* argument can be `'stdout`, in which case the same file-stream port or that is supplied as standard output is also used for standard error. For each port or `'stdout` that is provided, no pipe is created and the corresponding returned value is `#f`.

The caller of `dynamic-place*` is responsible for closing all returned ports; none are closed automatically.

The `dynamic-place*` procedure returns four values:

- a place descriptor value representing the created place;
- an output port piped to the place's standard input, or `#f` if *`in`* was a port;
- an input port piped from the place's standard output, or `#f` if *`out`* was a port;
- an input port piped from the place's standard error, or `#f` if *`err`* was a port or `'stdout`.

```
(place id body ...+)
```

creates a place that evaluates *`body`* expressions with *`id`* bound to a place channel. The *`body`*s close only over *`id`* plus the top-level bindings of the enclosing module, because the *`body`*s are lifted to a function that is exported by the module. The result of `place` is a place descriptor, like the result of `dynamic-place`.

```
(place* maybe-port ...
        id
        body ...+)

maybe-port =
           | #:in in-expr
           | #:out out-expr
           | #:err err-expr
```

like `place`, but supports optional `#:in`, `#:out`, and `#:err` expressions (at most one of each) to specify ports in the same way and with the same defaults as `dynamic-place*`. The result of a `place*` form is also the same as for `dynamic-place*`.

```
(place-wait p) → exact-integer?
  p : place?
```

returns the completion value of the place indicated by *p*, blocking until the place has terminated.

If any pumping threads were created to connect a nonfile-stream port to the ports in the place for *p* (see `dynamic-place`), `place-wait` returns only when the pumping threads have completed.

```
(place-dead-evt p) → evt?
  p : place?
```

returns a synchronizable event that is ready if and only if *p* has terminated.

If any pumping threads were created to connect a nonfile-stream port to the ports in the place for *p* (see `dynamic-place`), the event returned by `place-dead-evt` may become ready even if a pumping thread is still running.

```
(place-kill p) → void?
  p : place?
```

immediately terminates the place, setting the place's completion value to `1` if the place does not have a completion value already.

```
(place-break p [kind]) → void?
  p : place?
  kind : (or/c #f 'hang-up 'terminate) = #f
```

sends the main thread of place *p* a break.

```
(place-channel) → place-channel? place-channel?
```

returns two place channels. Data sent through the first channel can be received through the second channel, and data sent through the second channel can be received from the first.

Typically, one place channel is used by the current place to send messages to a destination place; the other place channel is sent to the destination place (via an existing place channel).

```
(place-channel-put pch v) → void
  pch : place-channel?
  v : place-message-allowed?
```

sends a message *v* on channel *pch*. Since place channels are asynchronous, `place-channel-put` calls are nonblocking.

See `place-message-allowed?` form information on automatic coercions in *v*, such as converting a mutable string to an immutable string.

```
(place-channel-get pch) → place-message-allowed?
  pch : place-channel?
```

returns a message received on channel *pch*, blocking until a message is available.

```
(place-channel-put/get pch v) → any/c
  pch : place-channel?
  v : any/c
```

sends an immutable message *v* on channel *pch* and then waits for a message (perhaps a reply) on the same channel.

```
(place-message-allowed? v) → boolean?
  v : any/c
```

returns `#t` if *v* is allowed as a message on a place channel, `#f` otherwise.

If `(place-enabled?)` returns `#f`, then the result is always `#t` and no conversions are performed on *v* as a message. Otherwise, the following kinds of data are allowed as messages:

- numbers, characters, booleans, and `#<void>`;
- symbols, where the `eq?`ness of uninterned symbols is preserved within a single message, but not across messages;
- strings and byte strings, where mutable strings and byte strings are automatically replaced by immutable variants;
- paths (for any platform);
- pairs, lists, vectors, and immutable prefab structures containing message-allowed values, where a mutable vector is automatically replaced by an immutable vector;
- hash tables where mutable hash tables are automatically replaced by immutable variants;
- place channels, where a place descriptor is automatically replaced by a plain place channel;
- file-stream ports and TCP ports, where the underlying representation (such as a file descriptor, socket, or handle) is duplicated and attached to a fresh port in the receiving place;
- C pointers as created or accessed via `ffi/unsafe`; and
- values produced by `shared-flvector`, `make-shared-flvector`, `shared-fxvector`, `make-shared-fxvector`, `shared-bytes`, and `make-shared-bytes`.

3.7 Performance Evaluation

I evaluated the performance of places by running the NASA Advanced Supercomputing (NAS) Parallel Benchmarks [4].[4] These benchmarks represent simplified kernels from computation fluid-dynamics problems. This section presents results for Racket,[5] Java, and Fortran/C versions of the NAS benchmarks.

I use two high-end workstations, Figure 3.9 that might be typical of a scientist's desktop machine. *Penghu* is a dual socket, quad-core per processor, Intel Xeon machine running Mac OS X. *Drdr* is a dual socket, hex-core per processor, AMD machine running Linux.

The NAS Parallel Benchmarks consists of seven benchmarks. Integer Sort (IS) is a simple histogram integer sort. Fourier Transform (FT) is a 3-D fast Fourier transform. FT computes three 1-D FFTs, one for each dimension. Conjugate Gradient (CG) approximates the smallest eigenvalue of a sparse unstructured matrix, which tests the efficiency of indirect memory access. MultiGrid (MG) solves a 3-D scalar Poisson equation and exercises memory transfers. Scalar Pentadiagonal (SP) is a 3-D Navier-Stokes solver using Beam-Warming approximate factorization. Block Tridiagonal (BT) is a Navier-Stokes solver using Alternating Direction Implicit approximate factorization. Lower and Upper (LU) is a Navier-Stokes solver using the symmetric successive over-relaxation method.

	Penghu	**Drdr**
OS	OS X 10.6.2	Ubuntu 10.4
Arch	x86_64	x86_64
Processor Type	Xeon	Opteron 2427
Processors	2	2
Total Cores	8	12
Clock Speed	2.8 GHz	2.2 GHz
L2 Cache	12MB	3MB
Memory	8 GB	16 GB
Bus Speed	1.6 GHz	1 GHz
Racket	v5.1.1.6	v5.1.1.6
gfortran	4.6.0 2010/7	4.4.3
Java	1.6.0_20	OpenJDK 1.6.0_18

Figure 3.9: Benchmark Machines

[4] `http://www.nas.nasa.gov/Resources/Software/npb.html`

[5] The Racket version of the NAS Parallel Benchmarks is available at `https://github.com/tewk/racketNAS`.

Each NAS benchmark consists of a range of problem size classes: from smallest to largest, they are S, W, A, B, and C. I ran the A size class on the shorter IS, FT, CG, MG benchmarks. On the longer benchmarks, SP, BT, and LU, I ran the W size class.

Each benchmark is represented by a row of graphs in Figures 3.10, 3.11, 3.12, 3.13, 3.14, and 3.15. The raw-performance graphs for each of the two benchmark machines comes first, followed by the speedup graphs. The raw-performance graph plots the number of threads versus the time to complete the benchmark with the left-most point (labelled "S") indicating the time for running the sequential benchmark without creating any places. The speedup graphs plot the number of threads versus the benchmark runtime divided by the benchmark time for one parallel thread. The gray line in the speed up graphs indicates perfect linear speedup.

In terms of raw performance, the Fortran/C implementation is the clear winner. Java comes in second in most benchmarks. Racket is third in most benchmarks, although it handily wins over Java in the SP and LU benchmarks.

More importantly, the Racket results demonstrate that our places implementation generally scales as well as the Java and Fortran/C versions do. In many of the benchmarks, running the Racket code with one parallel place takes only slightly longer than running the sequential code. The small difference in run times between sequential and one-place parallel versions suggests that the runtime cost of places for parallelization is practical.

The IS C result for the Penghu (Mac OS X) machine is uncharacteristically slower than the Java and Racket run times. The IS benchmark on the Drdr (Linux) machine is much faster. The NPB implementors wrote all the reference benchmarks in Fortran, except for IS. The NPB developers wrote the IS benchmark in C, using OpenMP's `threadprivate` directive. GCC versions prior to 4.6 refused to compile the IS benchmark under Mac OS X, emitting an error that `__threadlocal` was not supported. However, the prerelease GCC 4.6 successfully compiles and runs the IS benchmark. I believe that GCC 4.6 calls the `pthread_get_specific()` API function to implement OpenMP thread private variables, which increases the runtime of the IS implementation on Mac OS X.

The 3x difference in FT performance between Racket and Java is most likely due to Racket's lack of instruction-level scheduling and optimization. The negative scaling seen in the CG benchmark on Drdr for processor counts 7-12 is likely a chip locality issue when the computation requires both processor sockets. Unlike all the other benchmark kernels, the CG benchmark operates on a sparse matrix. The extra indirection in the sparse matrix representation reduces the effectiveness of memory caches and tests random memory accesses.

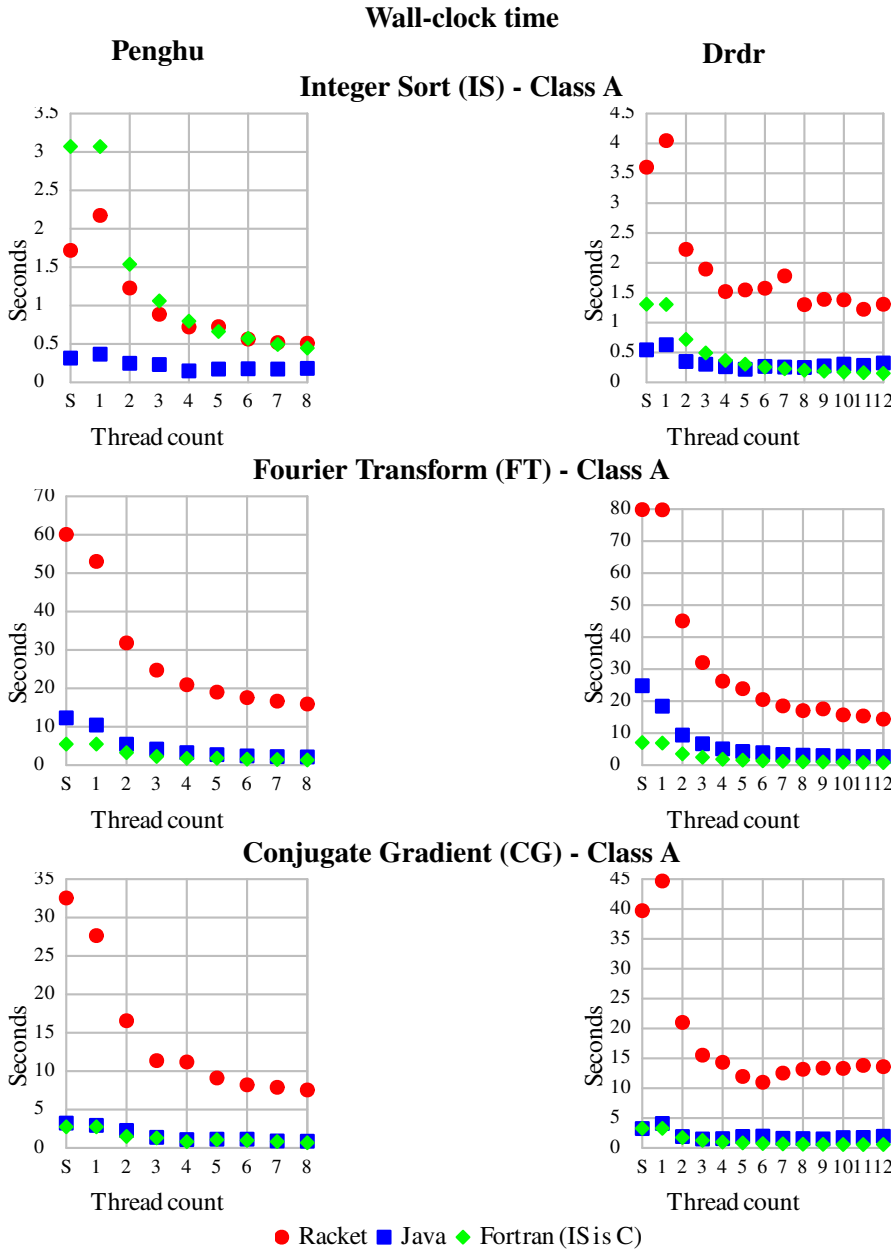

Figure 3.10: IS, FT, and CG wall-clock results

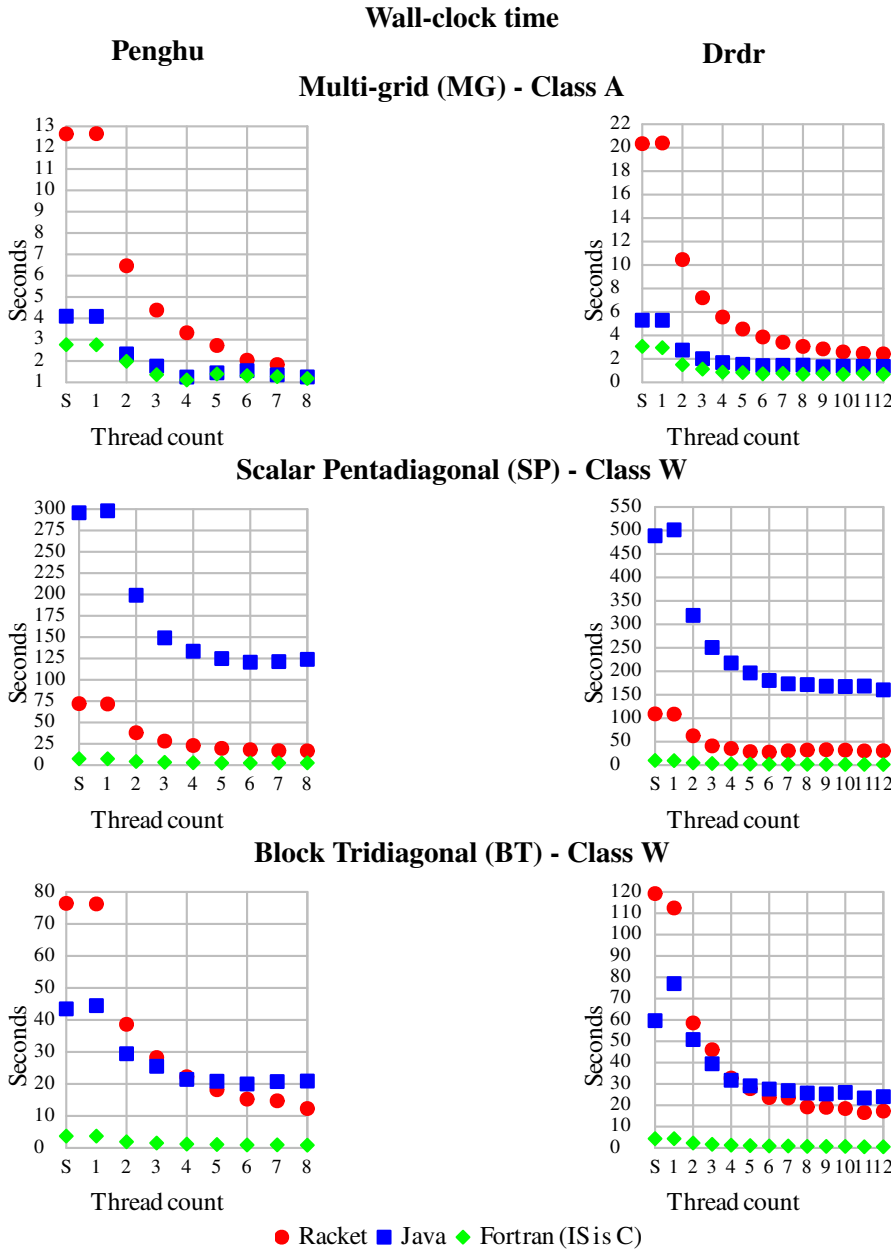

Figure 3.11: MG, SP, and BT wall-clock results

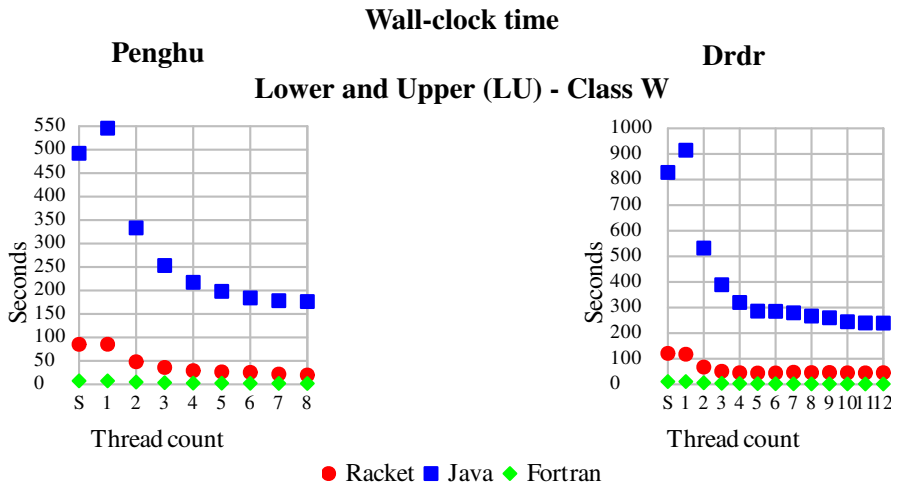

Figure 3.12: LU wall-clock results

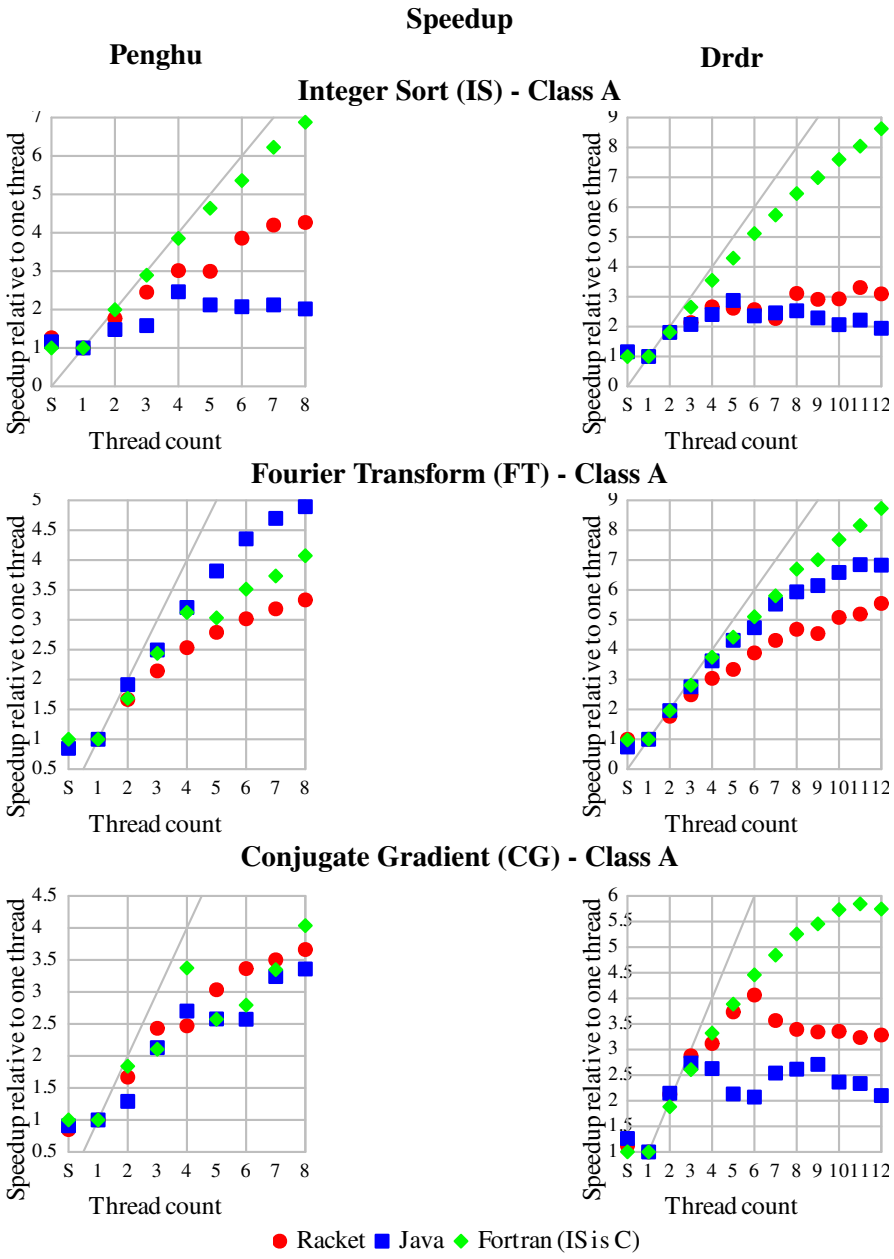

Figure 3.13: IS, FT, and CG speedup results

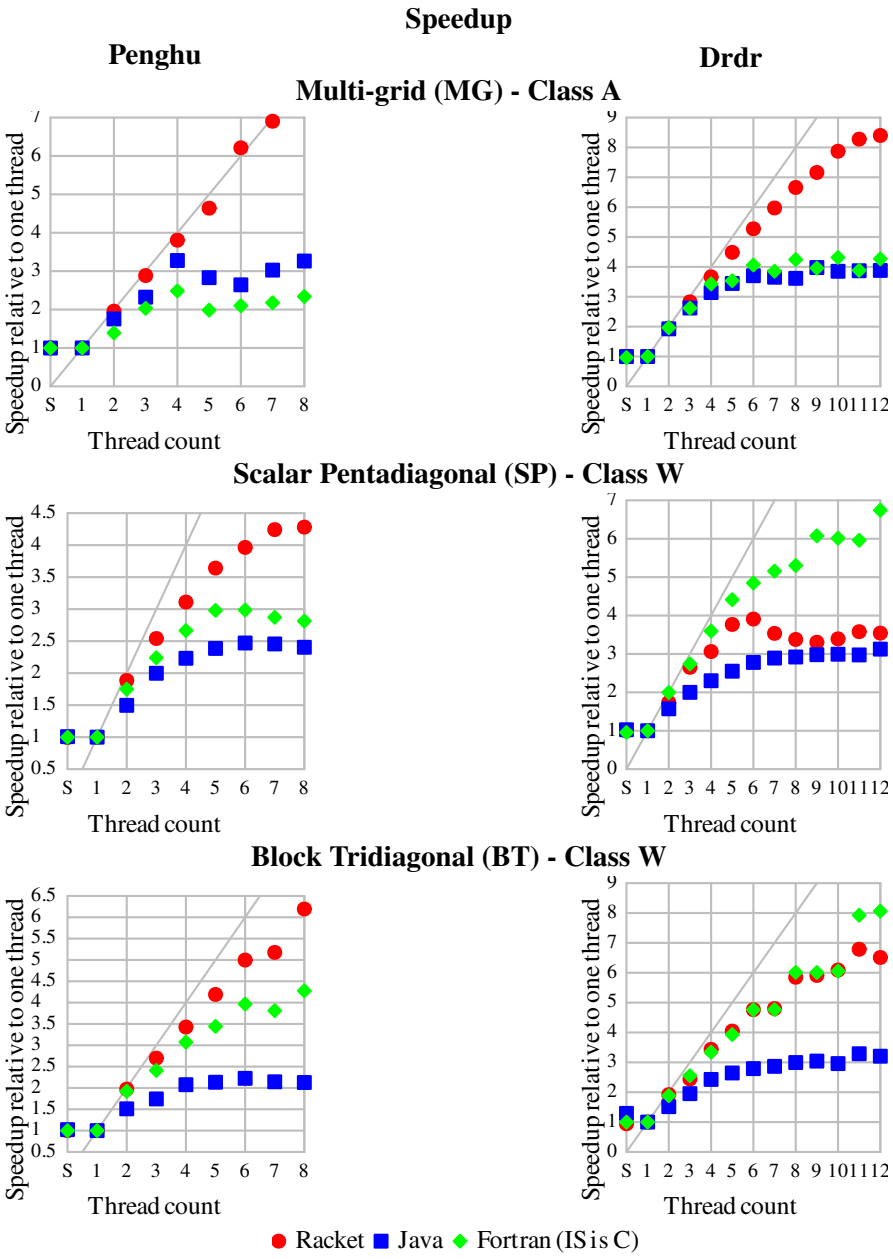

Figure 3.14: MG, SP, and BT speedup results

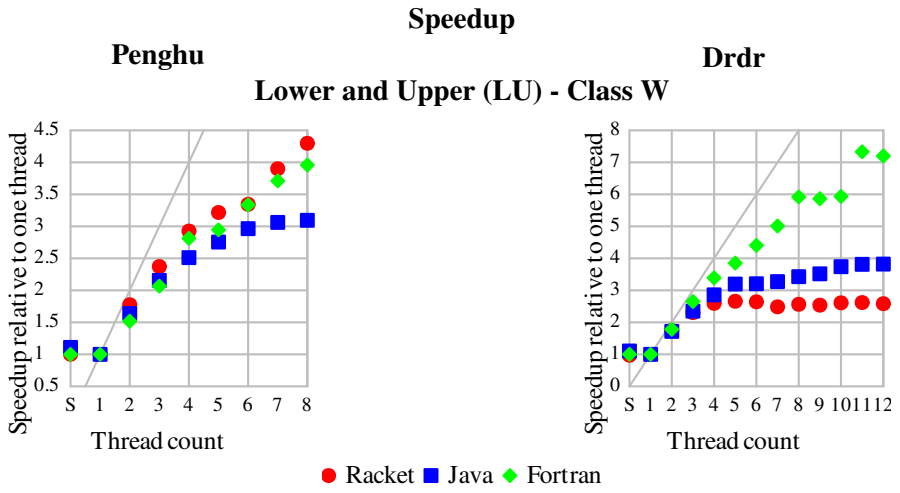

Figure 3.15: LU speedup results

The MG benchmark stresses a machine's memory subsystem in a different manner. During its computation, MG copies data back and forth between coarse and fine representations of its grid. On Mac OS X, I had to increase the Java maximum heap size from 128MB to 600MB for the MG benchmark to finish successfully. Java's maximum heap size on Linux appears to default to approximately 1/4th of the total system memory, which was sufficient for the MG benchmark to finish on our Linux test platform.

The 4x difference in runtimes between Java and Racket in SP and LU is most likely due to poor common subexpression elimination. While porting the Java benchmarks to Racket, I manually eliminated hundreds of common subexpressions by introducing local variables. The reference implementation's Fortran code has the same duplicated subexpressions as the Java version. In contrast to Java, the Fortran compiler appears to have a very effective subexpression elimination optimization pass.

When considering performance, one should take in account the programmer effort to achieve performance. Racket's state of the art macro facilities enable programmers to employ high levels of abstraction and eliminate highly parameterized, yet repeated code patterns. Table 3.1 compares programmer effort using the lines of code (LOC) metric. Racket handily wins the contest of small code size. Eliminating repeated code patterns with macros results in programs that more clearly identify the differences in the repeated patterns. Repeated code leads to cut and paste mistakes during authoring. When bugs are found, fixes must be repeated for each instance of repeated code. Languages which empower programmers to write concise, abstracted code, save programmer's time and grief.

Table 3.1: NAS Parallel Benchmarks Lines of Code

Benchmark	Racket LOC	Java LOC	Fortran LOC
IS	259	382	625
FT	414	900	615
CG	414	667	565
MG	587	1483	841
SP	1236	3825	2276
BT	1191	4099	3585
LU	1121	4555	3657

3.8 Conclusion

Places in Racket demonstrate how adding a message-passing layer to an existing runtime system can provide effective support for parallelism with a reasonable implementation effort. Our benchmark results demonstrate good scaling on traditional parallel tasks, and the use of places for parallel library compilation demonstrates that the implementation holds up in real-world use.

Although places are primarily designed for message-passing parallelism, shared mutable vectors of bytes, fixnums, or floating-point numbers are also supported; careful programmers may have good reasons to use these structures. Crucially, shared vectors of atomic data create few problems for the language implementation, so they are easily accommodated by the places API. Meanwhile, the Racket implementation is free to implement message-passing of immutable objects through sharing, if the trade-off in implementation complexity versus performance favors that direction, since sharing of immutable data is safe.

Places are a better model than the conventional "add threads; add locks until it stops crashing; remove locks until it scales better; repeat" approach to programming-language concurrency. Simply running the Racket test suite in multiple places uncovered the vast majority of bugs in our implementation. The same has not been true of our attempts to support concurrency with shared memory (e.g., with futures). Indeed, there seems to be no comparably simple way to find race conditions with threads and locks; many tools have been designed to help programmers find concurrency bugs—from Eraser [41] to GAMBIT [12]—but they suffer from problems with false positives, restrictions on supported code, problems scaling to large systems, or requiring assertions or other manual annotations. In contrast, bugs in the implementation of places were easy to find because they create permanently broken references (that are detected by the garbage collector) rather than fleeting timing effects.

CHAPTER 4

DISTRIBUTED PLACES

4.1 Introduction

Dynamic, functional languages are important as rapid development platforms for solving everyday problems and completing tasks. As programmers embrace parallelism in dynamic programming languages, the need arises to extend multicore parallelism to multinode parallelism. Distributed places delivers multinode parallelism to Racket by building on top of the existing places [47] infrastructure.

The right extensions to dynamic, functional languages enable the introduction of a hierarchy of parallel programming abstractions. Language extension allows these parallel programming abstractions to be concisely mapped to different hardware such as a shared memory node or a distributed memory machine. Distributed places are not an add-on library or a foreign function interface (FFI). Instead, Racket's places and distributed places are language extensions on which higher-level distributed programming frameworks can easily be expressed. An RPC mechanism, map reduce, MPI, and nested-data parallelism are all concisely and easily built on top of distributed places. These higher-level frameworks meld with the Racket language to create extended languages, which describe different types of distributed programming.

Making distributed places integrate seamlessly into a dynamic language requires a few key steps. First, the language needs to implement the safe, robust, and scalable parallelism solution which the places design provides. Second, the places API must be extended to support spawning places on remote machines. Place channels also have to be extended to allow transparent internode communication over an underlying sockets layer. Third, the language's concurrency system must be extensible and include new distributed places events. As an example, Racket distributed places and place channels are events which can be waited on concurrently with other Racket event objects such as file ports, sockets, threads, channels, etc. Together, places and distributed places form a foundation capable of supporting higher-level parallel frameworks.

4.2 Design

Distributed places' design orginates from the design of places. Places is Racket's message-passing form of parallelism. The isolated execution of places avoids the typical interference problems of threads executing in a single address space. Instead, places positions the programmer to think about the data-placement and communication needs of a parallel program to enable sustained scalability. As a program moves from multicore parallelism to multinode parallelism latency increases and bandwidth decreases. Data-placement and communication patterns become even more crucial.

Much of a distributed programming API is littered with system administration tasks that impede programmers from focusing on programming and solving problems. First, programmers have to authenticate and launch their programs on each node in the distributed system. Then they have to establish communication links between the nodes in the system, before they can begin working on the problem itself. The work of the distributed places framework is to provide support for handling the problems of program launch and communication link establishment.

Racket's distributed places API design is centered around machine nodes that do computation in places. The user/programmer configures a new distributed system using declarative syntax and callbacks. By specifying a hostname and port number, a programmer can launch a new place on a remote host. In the simplest distributed-places programs, hostnames and port numbers are hard-wired. When programmers need more control, distributed places permits complete programmatic configuration of node launch and communication link parameters.

The hello world example in Figure 4.1 demonstrates the key components of a places program. Appearing first, the `hello-world` procedure is called to create hello-world places. The `main` module follows and contains the code to construct and communicate with a `hello-world` place.

Looking closer at the `main` module, the `hello-world` place is created using `dynamic-place`.

```
(dynamic-place module-path start-proc) → place?
  module-path : module-path?
  start-proc : symbol?
```

The `dynamic-place` procedure creates a place to run the procedure that is identified by *`module-path`* and *`start-proc`*. The result is a place descriptor value that represents the new parallel task; the place descriptor is returned immediately. The place descriptor is also a place channel to initiate communication between the new place and the creating place.

The module indicated by *`module-path`* must export a function with the name *`start-proc`*. The exported function must accept a single argument, which is a place channel that corresponds to the other end of communication for the place channel that is returned by `dynamic-place`.

```
1 #lang racket/base
2 (require racket/place
3          racket/place/distributed)
4
5 (provide hello-world)
6
7 (define (hello-world ch)
8   (printf/f "hello-world received: ~a\n"
9             (place-channel-get ch))
10  (place-channel-put ch "Hello World\n")
11  (printf/f "hello-world sent: Hello World\n"))
12
13 (module+ main
14  (define p (dynamic-place (quote-module-path "..")
15                           'hello-world))
16
17  (place-channel-put p "Hello")
18  (printf/f "main received: ~a\n"
19            (place-channel-get p))
20  (place-wait p))
```

Figure 4.1: Place's Hello World

The `(quote-module-path "..")` and `'hello-world` arguments on lines 17 and 18 of Figure 4.2 specify the procedure address of the new place to be launched on the remote node. In this example, the `(quote-module-path "..")` argument provides the module path to the parent module of `main`, where the `'hello-world` procedure is located.

```
13 (module+ main
14  (define n (create-place-node "host2"
15                               #:listen-port 6344))
16  (define p (dynamic-place #:at n
17                           (quote-module-path "..")
18                           'hello-world))
19  ...)
```

Figure 4.2: Distributed Hello World

Places communicate over place channels which allow structured data communication between places. Supported structured data includes booleans, numbers, characters, symbols, byte strings, Unicode strings, filesystem paths, pairs, lists, vectors, and "prefab" structures (i.e., structures that are transparent and whose types are universally named). Place channels themselves can be sent in messages across place channels, so that communication is not limited to the creator of a place and its children places; by sending place channels as messages, a program can construct custom message topologies.

```
(place-channel-put ch v) → void?
  ch : place-channel?
  v : place-message-allowed?
(place-channel-get ch) → place-message-allowed?
  ch : place-channel?
```

The `place-channel-put` function asynchronously sends a message *v* on channel *ch* and returns immediately. The `place-channel-get` function waits until a message is available from the place channel *ch*.

```
(place-wait p) → void?
  p : place?
```

Finally, the `place-wait` procedure blocks until *p* terminates.

The distributed hello world example in Figure 4.2 shows the two differences between a simple places program and a simple distributed places program. The `create-place-node` procedure uses `ssh` to start a new remote node on `host2` and assumes that ssh is configurered correctly. Upon launch, the remote node listens on port `6344` for incoming connections. Once the remote node is launched, a TCP connection to the new node is established. The `create-place-node` returns a node descriptor object, `n`, which allows for administration of the remote node. The remote place is created using `dynamic-place`. The new `#:at` keyword argument specifies the node on which to launch the new place.

Remotely spawned places are private. Only the node that spawned the place can communicate with it through its descriptor object. Named places allow programmers to make a distributed place publicly accessible. Named places are labeled with a name when they are created.

```
(define p (dynamic-place #:at n
                         #:named 'helloworld1
                         (quote-module-path "..")
                         'hello-world))
```

Any node can connected to a named place by specifying the node and name to connect to.

```
(connect-to-named-place node 'helloworld1)
```

4.3 Higher-Level APIs

The distributed places implementation is a foundation that can support a variety of higher-level APIs and parallel processing frameworks such as Remote Procedure Calls (RPC), Message Passing Interface (MPI) [30], MapReduce [13], and Nested Data Parallelism [8]. All of these higher-level APIs and frameworks are built on top of named places.

4.3.1 RPC via Named Places

Named places make a place's interface public at a well-known address: the host, port, and name of the place. They provide distributed places with a form of computation similar to the actor model [24]. Using named places and the `define-named-remote-server` form, programmers can build distributed places that act as remote procedure call (RPC) servers. The example in Figure 4.3 demonstrates how to launch a remote Racket node instance, launch a remote procedure call (RPC) tuple server on the new remote node instance, and start a local event loop that interacts with the remote tuple server.

The `create-place-node` procedure in Figure 4.3 connects to `"host2"` and starts a distributed place node there that listens on port 6344 for further instructions. The descriptor to the new distributed place node is assigned to the `remote-node` variable. Next, the `dynamic-place` procedure creates a new named place on the `remote-node`. The named place will be identified in the future by its name symbol `'tuple-server`.

The code in Figure 4.4 contains the use of the `define-named-remote-server` form, which defines a RPC server suitable for invocation by `dynamic-place`. The RPC `tuple-server` allows for named tuples to be stored into a server-side hash table and later retrieved. It also demonstrates one-way "cast" procedures, such as `hello`, that do not return a value to the remote caller.

For the purpose of explaining the `tuple-server` implementation, Figure 4.5 shows the macro expansion of the RPC tuple server. Typical users of distributed places do not need to understand the expanded code to use the `define-named-remote-server` macro. The `define-named-remote-server` form, in Figure 4.5, takes an identifier and a list of custom expressions as its arguments. A place function is created by prepending the `make-` prefix to the identifier `tuple-server`. The `make-tuple-server` identifier is the symbol given to the `dynamic-place` form

```
1 #lang racket/base
2 (require racket/place/distributed
3          racket/class
4          racket/place
5          racket/runtime-path
6          "tuple.rkt")
7 (define-runtime-path tuple-path "tuple.rkt")
8
9 (module+ main
10   (define remote-node (create-place-node
11                        "host2"
12                        #:listen-port 6344))
13   (define tuple-place (dynamic-place
14                        #:at remote-node
15                        #:named 'tuple-server
16                        tuple-path
17                        'make-tuple-server))
18
19   (define c (connect-to-named-place remote-node
20                                     'tuple-server))
21   (define d (connect-to-named-place remote-node
22                                     'tuple-server))
23   (tuple-server-hello c)
24   (tuple-server-hello d)
25   (displayln (tuple-server-set c "user0" 100))
26   (displayln (tuple-server-set d "user2" 200))
27   (displayln (tuple-server-get c "user0"))
28   (displayln (tuple-server-get d "user2"))
29   (displayln (tuple-server-get d "user0"))
30   (displayln (tuple-server-get c "user2")))
```

Figure 4.3: Tuple RPC Example

in Figure 4.3. The `define-state` custom form translates into a simple `define` form, which is closed over by the `define-rpc` forms.

The `define-rpc` form is expanded into two parts. The first part is the client stubs that call the RPC functions. The stubs can be seen at the top of Figure 4.5. The client function name is formed by concatenating the `define-named-remote-server` identifier, `tuple-server`, with the RPC function name, `set`, to form `tuple-server-set`. The RPC client functions take a destination argument which is a `remote-connection%` descriptor followed by the RPC function's arguments. The RPC client function sends the RPC function name, `set`, and the RPC arguments to the destination by calling an internal function `named-place-channel-put`. The RPC client then

```
1 #lang racket/base
2 (require racket/match
3          racket/place/define-remote-server)
4
5 (define-named-remote-server tuple-server
6
7   (define-state h (make-hash))
8   (define-rpc (set k v)
9     (hash-set! h k v)
10    v)
11  (define-rpc (get k)
12    (hash-ref h k #f))
13  (define-cast (hello)
14    (printf "Hello from define-cast\n")
15    (flush-output)))
```

Figure 4.4: Tuple Server

calls `named-place-channel-get` to wait for the RPC response.

The second part of the expansion part of `define-rpc` is the server implementation of the RPC call. The server is implemented by a match expression inside the `make-tuple-server` function. Messages to named places are placed as the first element of a list where the second element is the source or return channel on which to respond. For example, in `(list (list 'set k v) src)` the inner list is the message while `src` is the place-channel to send the reply on. The match clause for `tuple-server-set` matches on messages beginning with the `'set` symbol. The server executes the RPC call with the communicated arguments and sends the result back to the RPC client. The `define-cast` form is similar to the `define-rpc` form except there is no reply message from the server to client.

The named place, shown in the tuple server example, follows an actor-like model by receiving messages, modifying state, and sending responses. Racket macros enables the easy construction of RPC functionality on top of named places.

4.3.2 Racket Message Passing Interface

RMPI is Racket's implementation of the basic MPI operations. A RMPI program begins with the invocation of the `rmpi-launch` procedure, which takes two arguments. The first is a hash from racket keywords to values of default configuration options. The `rmpi-build-default-config` helper procedure takes a list of Racket keyword arguments and forms the hash of optional

```
(module named-place-expanded racket/base
   (require racket/place racket/match)
   (define/provide
    (tuple-server-set dest k v)
     (named-place-channel-put dest (list 'set k v))
     (named-place-channel-get dest))
   (define/provide
    (tuple-server-get dest k)
     (named-place-channel-put dest (list 'get k))
     (named-place-channel-get dest))
   (define/provide
    (tuple-server-hello dest)
     (named-place-channel-put dest (list 'hello)))
   (define/provide
    (make-tuple-server ch)
     (let ()
       (define h (make-hash))
       (let loop ()
         (define msg (place-channel-get ch))
          (match
           msg
           ((list (list 'set k v) src)
            (define result (let ()
                             (hash-set! h k v)
                             v))
            (place-channel-put src result)
            (loop))
           ((list (list 'get k) src)
            (define result (let ()
                             (hash-ref h k #f)))
            (place-channel-put src result)
            (loop))
           ((list (list 'hello) src)
            (define result
              (let ()
                (printf
                  "Hello from define-cast\n")
                (flush-output)))
            (loop)))
         loop)))
   (void))
```

Figure 4.5: Macro Expansion of Tuple Server

configuration values. The second argument is a list of configurations, one for each node in the distributed system. A configuration is made up of a hostname, a port, a unique name, a numerical RMPI process id, and an optional hash of additional configuration options. An example of `rmpi-launch` follows.

```
(rmpi-launch
  (rmpi-build-default-config
    #:racket-path "/tmp/mplt/bin/racket"
    #:distributed-launch-path
      (build-distributed-launch-path
        "/tmp/mplt/collects")
    #:rmpi-module "/tmp/mplt/kmeans.rkt"
    #:rmpi-func 'kmeans-place
    #:rmpi-args
      (list "/tmp/mplt/color100.bin"
            #t 100 9 10 0.0000001))

  (list (list "n1.example.com" 6340 'kmeans_0 0)
        (list "n2.example.com" 6340 'kmeans_1 1)
        (list "n3.example.com" 6340 'kmeans_2 2)
        (list "n4.example.com" 6340 'kmeans_3 3
              (rmpi-build-default-config
                #:racket-path "/bin/racket"))))
```

The `rmpi-launch` procedure spawns the remote nodes first and then spawns the remote places named with the unique name from the config structure. After the nodes and places are spawned, `rmpi-launch` sends each spawned place its RMPI process id, the config information for establishing connections to the other RMPI processes, and the initial arguments for the RMPI program. The last function of `rmpi-launch` is to rendezvous with RMPI process 0 when it calls `rmpi-finish` at the end of the RMPI program.

The `rmpi-init` procedure is the first call that should occur inside the `#:rmpi-func` place procedure. The `rmpi-init` procedure takes one argument `ch`, which is the initial `place-channel` passed to the `#:rmpi-func` procedure. The `rmpi-init` procedure communicates with `rmpi-launch` over this channel to receive its RMPI process id and the initial arguments for the RMPI program.

```
(define (kmeans-place ch)
  (define-values (comm args tc) rmpi-init ch)
  ;;; kmeans rmpi computation ...
  (rmpi-finish comm tc))
```

The `rmpi-init` procedure has three return values: an opaque communication structure which is passed to other RMPI calls, the list of initial arguments to the RMPI program, and a typed channel wrapper for the initial place-channel it was given. The typed channel wrapper allows for the out of order reception of messages. Messages are lists and their type is the first item of the list, which must

be a racket symbol. A typed channel returns the first message received on the wrapped channel that has the type requested. Messages of other types that are received are queued for later requests.

The `rmpi-comm` structure, returned by `rmpi-init`, is the communicator descriptor used by all other RMPI procedures. The RMPI informational functions `rmpi-id` and `rmpi-cnt` return the current RMPI process id and the total count of RMPI processes, respectively.

```
> (rmpi-id comm)
3



> (rmpi-cnt comm)
8
```

The `rmpi-send` and `rmpi-recv` procedures provide point-to-point communication between two RMPI processes.

```
> (rmpi-send comm dest-id '(msg-type1 "Hi"))



> (rmpi-recv comm src-id)
'(msg-type1 "Hi")
```

With the `rmpi-comm` structure, the programmer can also use any of the RMPI collective procedures: `rmpi-broadcast`, `rmpi-reduce`, `rmpi-allreduce`, or `rmpi-barrier` to communicate values between the nodes in the RMPI system.

The `(rmpi-broadcast comm 1 (list 'a 12 "foo"))` expression broadcasts the list `(list 'a 12 "foo")` from RMPI process `1` to all the other RMPI processes in the `comm` communication group. Processes receiving the broadcast execute `(rmpi-broadcast comm 1)` without specifying the value to send. The `(rmpi-reduce comm 3 + 3.45)` expression does the opposite of broadcast by reducing the local value `3.45` and all the other procesess local values to RMPI process `3` using the `+` procedure to do the reduction. The `rmpi-allreduce` expression is similar to `rmpi-reduce` except that the final reduced value is broadcasted to all processes in the system after the reduction is complete. Synchronization among all the RMPI processes occurs through the use of the `(rmpi-barrier comm)` expression, which is implemented internally using a simple reduction followed by a broadcast.

Distributed places are simply computation resources connected by socket communications. This simple design matches MPI's model and makes RMPI's implementation very natural. The

RMPI layer demonstrates how distributed places can provide the foundations of other distributed programming frameworks such as MPI.

4.3.3 Map Reduce

Our MapReduce implementation is patterned after the Hadoop [1] framework. Key value pairs are the core data structures that pass through the map and reduce stages of the computation. In the following example, the number of word occurrences is counted across a list of text files. The files have been preprocessed so that there is only one word per line.

Figure 4.6 shows the different actors in the MapReduce paradigm. The program node `P` creates the MapReduce workers group. When a `map-reduce` call is made, the program node serves as the controller of the worker group. It dispatches mapper tasks to each node and waits for them to respond as finished with the mapping task. Once a node has finished its mapping task, it runs the reduce operation on its local data. Given two nodes in the reduced state, one node can reduce to the other freeing one node to return to the worker pool for allocation to future tasks. Once all the nodes have reduced to a single node, the `map-reduce` call returns the final list of reduced key values.

The first step in using distributed place's MapReduce implementation is to create a list of worker nodes. This is done by calling the `make-map-reduce-workers` procedure with a list of hostnames and ports to launch nodes at.

```
(define config (list (list "host2" 6430)
                     (list "host3" 6430)))
(define workers (make-map-reduce-workers config))
```

Once a list of worker nodes have been spawned, the programmer can call `map-reduce` supplying the list of worker nodes, the config list, the procedure address of the mapper, the procedure address of the reducer, and a procedure address of an optional result output procedure. Procedure addresses are lists consisting of the quoted-module-path and the symbol name of the procedure being addressed.

```
(map-reduce workers config tasks
            (list (quote-module-path "..") 'mapper)
            (list (quote-module-path "..") 'reducer)
            #:outputer (list (quote-module-path "..")
                             'outputer))
```

Tasks can be any list of key value pairs. In this example, the keys are the task numbers and the values are the input files the mappers should process.

```
(define tasks (list (list (cons 0 "/tmp/w0"))
                    (list (cons 1 "/tmp/w1"))
                    ...))
```

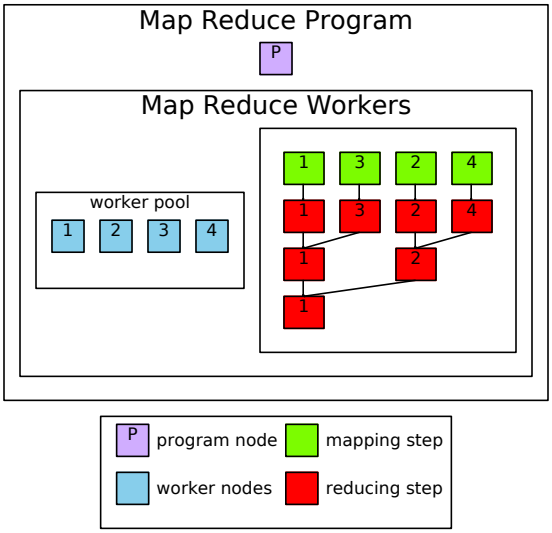

Figure 4.6: MapReduce Program

The mapper procedure takes a list of key value pairs as its argument and returns the result of the map operation as a new list of key value pairs. The input to the mapper, in this example, is a list of a single pair containing the task number and the text file to process, `(list (cons 1 "w0.txt"))`. The output of the mapper is a list of each word in the file paired with 1, its initial count. Repeated words in the text are repeated in the mappers output list. Reduction happens in the next step.

```
;;(->
;;  (listof (cons any any))
;;  (listof (cons any any)))
(define/provide (mapper kvs)
  (for/first ([kv kvs])
    (match kv
      [(cons k v)
       (with-input-from-file
         v
         (lambda ()
           (let loop ([result null])
             (define l (read-line))
             (if (eof-object? l)
                 result
                 (loop (cons (cons l 1)
                             result))))))])))
```

After a task has been mapped, the MapReduce framework sorts the output key value pairs by key. The framework also coalesces pairs of key values with the same key into a single pair of the key and the list of values. As an example, the framework transforms the output of the mapper

`'(("house" 1) ("car" 1) ("house" 1))` into `'(("car" (1)) ("house" (1 1)))`

The reducer procedure takes, as input, this list of pairs, where each pair consists of a key and a list of values. For each key, the reducer reduces the list of values to a list of a single value. In the word count example, an input pair, `(cons "house" '(1 1 1 1))` will be transformed to `(cons "house" '(4))` by the reduction step.

```
;;(->
;; (listof (cons any (listof any)))
;; (listof (cons any (listof any))))
(define/provide (reducer kvs)
  (for/list ([kv kvs])
    (match kv
      [(cons k v)
       (cons k (list (for/fold ([sum 0])
                               ([x v])
                       (+ sum x))))])))
```

Once each mapped task has been reduced, the outputs of the reduce steps are further reduced until a single list of word counts remains. Finally, an optional output procedure is called which prints out a list of words and their occurrence count and returns the total count of all words.

```
(define/provide (outputer kvs)
  (displayln
    (for/fold ([sum 0]) ([kv kvs])
      (printf "~a - ~a\n" (car kv) (cadr kv))
      (+ sum (cadr kv)))))
```

4.3.4 Nested Data Parallelism

The last parallel processing paradigm implemented on top of distributed places is nested data parallelism [22]. In this paradigm, recursive procedure calls create subproblems that can be parallelized. An implementation of parallel quicksort demonstrates nested data parallelism built on top of distributed places.

The distributed places, nested data parallelism API – `ndp-get-node`, `ndp-sendwork`, `ndp-get-result`, and `ndp-return-node` – is built on top of the RMPI layer. The main program node, depicted as `P` in Figure 4.7, creates the `ndp-group`. The `ndp-group` consists of a coordinating node, `0`, and a pool of worker nodes `1, 2, 3, 4`. The coordinating node receives a sort request from `ndp-sort` and forwards the request to the first available worker node, node `1`. Node `1` divides the input list in half and requests a new node from the coordinator to process the second half of the input. The yellow bars on the right side of Figure 4.7 show the progression as the sort input is subdivided and new nodes are requested from the coordinator node. Once the sort is complete, the result is returned to the coordinator node, which returns the result to the calling program `P`.

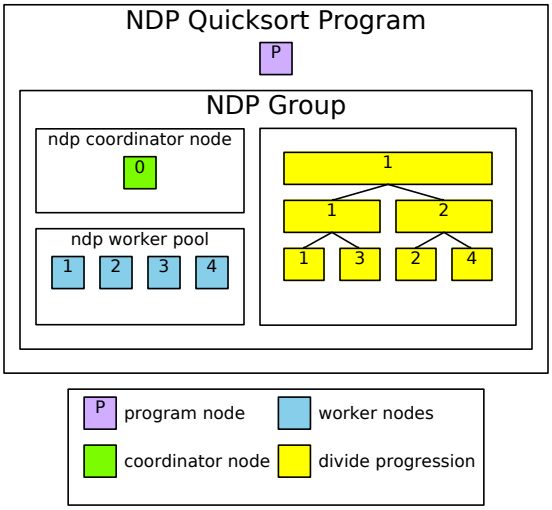

Figure 4.7: NDP Program

Like the previous two examples, the nested data parallel quicksort example begins by spawning a group of worker processes.

```
(define config
  (list (list "host2" 6340)
        (list "host3" 6340)
        (list "host4" 6340)
        (list "host5" 6340)
        (list "host6" 6340)))

(define ndp-group (make-ndp-group config))
```

Next, the sort is performed by calling `ndp-qsort`.

```
(displayln (ndp-qsort (list 9 1 2 8 3 7 4 6 5 10)
                      ndp-config))
```

The `ndp-qsort` procedure is a stub that sends the procedure address for the `ndp-parallel-qsort` procedure and the list to sort to the `ndp-group`. The work of the parallel sort occurs in the `ndp-parallel-sort` procedure. First, the `partit` procedure picks a pivot and partitions the input list into three segments: less than the pivot, equal to the pivot, and greater than the pivot. If a worker node can be obtained from the `ndp-group` by calling `ndp-get-node`, the `gt` partition is sent to the newly obtained worker node to be recursively sorted. If all the worker nodes are taken, the `gt` partition is sorted locally using the `ndp-serial-qsort` procedure. Once the `lt` partition is sorted recursively on the current node, the `gt-part` is checked to see if it was computed locally or dispatched to a remote node. If the part was dispatched to a remote node, its results are retrieved

from the remote node by calling `ndp-get-result`. After the results are obtained, the remote node `node` can be returned to the `ndp-group` for later use. Finally, the sorted parts are appended to form the final sorted list result.

```
(define (ndp-parallel-qsort l ndp-group)
  (cond
    [(< (length l) 2) l]
    [else
      (define-values (lt eq gt) (partit l))

      ;; spawn off gt partition
      (define gt-ref
         (define node (ndp-get-node ndp-group))
         (cond
           [node
             (cons #t (ndp-send-work
                        ndp-group
                        node
                        (list (quote-module-path)
                              'ndp-parallel-qsort)
                        gt))]
           [else
             (cons #f (ndp-serial-qsort gt))]))

      ;; compute lt partition locally
      (define lt-part
        (ndp-parallel-qsort lt ndp-group))

      ;; retrieve remote results
      (define gt-part
        (match gt-ref
          [(cons #t node-id)
            (begin0
              (ndp-get-result ndp-group node-id)
              (ndp-return-node ndp-group node-id))]
          [(cons #f part) part]))

      (append lt-part eq gt-part)]))
```

4.4 Implementation

A key part of the distributed place implementation is that distributed places is a layer over places, and parts of the places layer are exposed through the distributed places layer. In particular, each node, in Figure 4.8, begins life with one initial place, the message router. The message router listens on a TCP port for incoming connections from other nodes in the distributed system. The message router serves two primary purposes: it multiplexes place messages and events on TCP connections between nodes and it services remote spawn requests for new places.

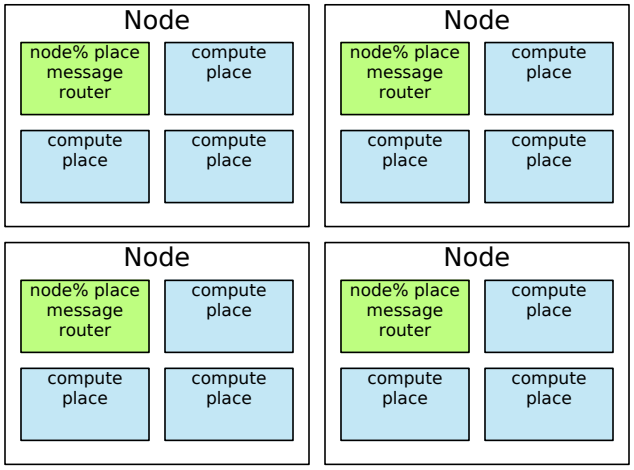

Figure 4.8: Distributed Places Nodes

There are a variety of distributed places commands which spawn remote nodes and places. These command procedures return descriptor objects for the nodes and places they create. The descriptor objects allow commands and messages to be communicated to the remote controlled objects. In Figure 4.9, when node A spawns a new node B, A is given a `remote-node%` object with which to control B. Consequently, B is created with a `node%` object that is connected to A's `remote-node%` descriptor via a TCP socket connection. B's `node%` object is the message router for the new node B. A can then use its `remote-node%` descriptor to spawn a new place on node B. Upon successful spawning of the new place on B, A is returned a `remote-place%` descriptor object. On node B, a `place%` object representing the newly spawned place is attached to B's `node%` message-router. The `remote-connection%` descriptor object represents a connection to a named place. At the remote node, B, a `connection%` object intermediates between the `remote-connection%` and its destination named-place.

To communicate with remote nodes, a place message must be serializable. As a message-passing implementation, places send a copy of the original message when communicating with other places. Thus, the content of a place message is inherently serializable and transportable between nodes of a distributed system.

To make place channels distributed, `place-socket-bridge%` proxies need to be created under the hood. The `place-socket-bridge%`s listen on local place channels and forward place messages over TCP sockets to remote place channels. Each node in a Racket distributed system must either explicitly pump distributed messages by registering each proxy with `sync` or bulk register the

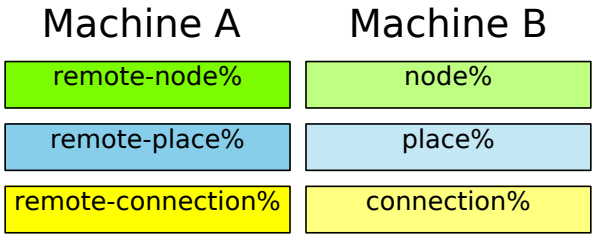

Figure 4.9: Descriptor (Controller) - Controlled Pairs

proxies, via the `remote-node%` descriptor, with a message router which can handle the pumping in a background thread.

Figure 4.10 shows the layout of the internal objects in a simple three node distributed system. The node at the top of the figure is the original node spawned by the user. Early in the instantiation of the top node, two additional nodes are spawned, node 1 and node 2. Then two places are spawned on each of node 1 and node 2. The instantiation code of the top node ends with a call to the `message-router` form. The `message-router` contains the `remote-node%` instances and the `after-seconds` and `every-seconds` event responders. Event responders execute when specific events occur, such as a timer event, or when messages arrive from remote nodes. The message router de-multiplexes events and place messages from remote nodes and dispatches them to the correct event responder.

Finally, function overloading is used to allow `place-` functions, such as `place-channel-get`, `place-channel-put`, and `place-wait`, to operate transparently on both place and distributed place instances. To accomplish this, distributed place descriptor objects are tagged as implementing the `place<%>` interface using a Racket structure property. Then `place-` functions dynamically dispatch to the distributed place version of the function for distributed place instances or execute the original function body for place instances.

4.5 Distributed Places Performance

Two of the NAS Parallel Benchmarks, IS and CG, are used to test the performance of the Racket distributed places implementation. The Fortran/C MPI version of the benchmarks were ported to Racket's distributed places. Performance testing occurred on 8 quad-core Intel i7 920 machines. Each machine was equipped with at least 4 gigabytes of memory and a 1 gigabit Ethernet connection.

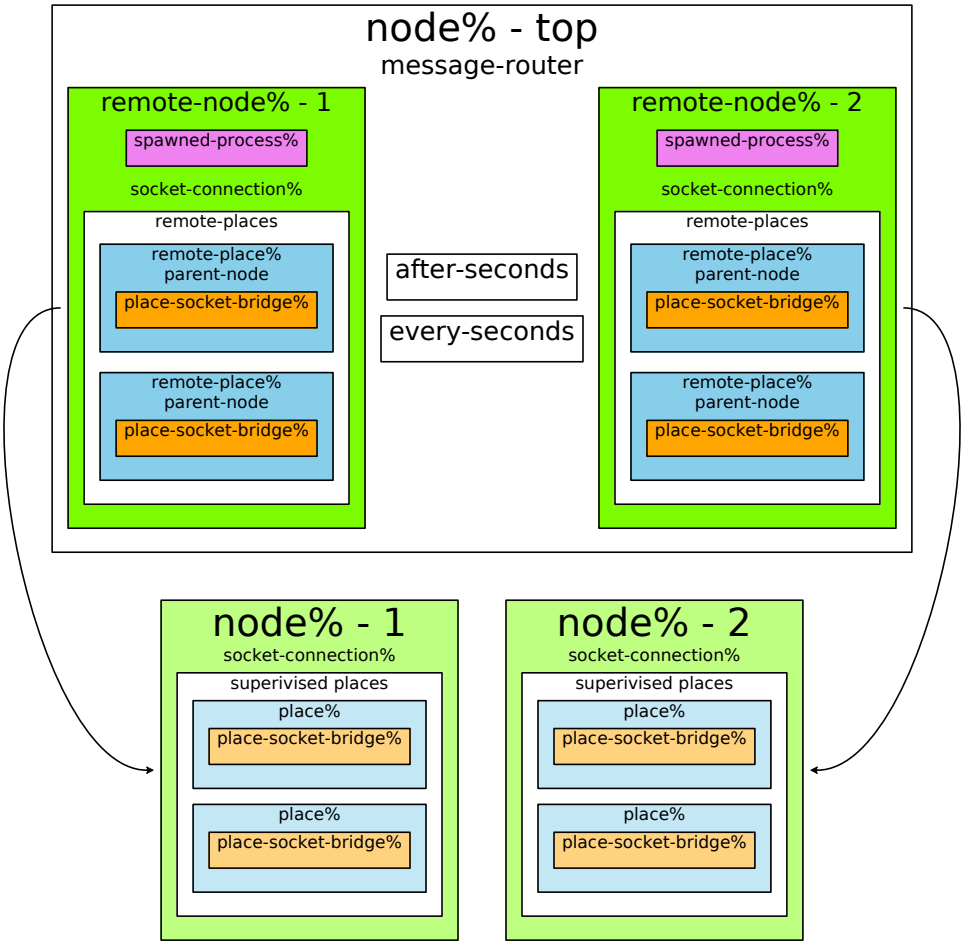

Figure 4.10: Three Node Distributed System

Performance numbers are reported for both Racket and Fortran/C versions of the benchmarks in Figure 4.11 and Figure 4.12, respectively. Racket's computational times scaled appropriately as additional nodes were added to the distributed system. Computational times are broken out and graphed in isolation to make computational scaling easier to see.

Racket communication times were larger than expected. There are several factors, stacked on top of one another, that explain the large communication numbers. First, five copies of the message occur during transit from source to destination. In a typical operation, a segment of a large flonum vector needs to be copied to a destination distributed place. The segment is copied out of the large flonum vector into a new flonum vector message. The message vector's length is the length of the segment to be sent. Next, the newly constructed vector message is copied over a place channel from the computational place to the main thread which serializes the message out a TCP socket to its destination. When the message arrives at its destination node, the message is deserialized and copied a fourth time over a place channel to the destination computational place. Finally, the elements of the message vector are copied into the mutable destination vector.

Racket's MPI implementation, RMPI, is not as sophisticated as the standard MPICH [31] implementation. MPICH has nonblocking sends and receives that allow messages to flow both directions simultaneously. Both the NAS Parallel Benchmarks used, IS and CG, use nonblocking MPI receives. RMPI on the other hand, always follows the typical protocol design of sending data in one direction and then receiving data from the opposite direction.

The largest contributor to Racket's excessive communication times is the serialization costs of the Racket primitive `write`. On Linux, serialization times are two orders of magnitude larger than the time to write raw buffers. One solution would be to replace distributed place's communication subsystem with FFI calls to an external MPI library. This solution would bypass the expensive `write` calls currently used in distributed places. Another viable solution would be to recognize messages that are vectors of flonums and use a restricted-form of write that could write flonum vectors as efficiently as raw buffers. Finally, it should be noted that using Racket's `write` is advantageous in cases where the message to be sent is a complex object graph instead of a simple raw buffer.

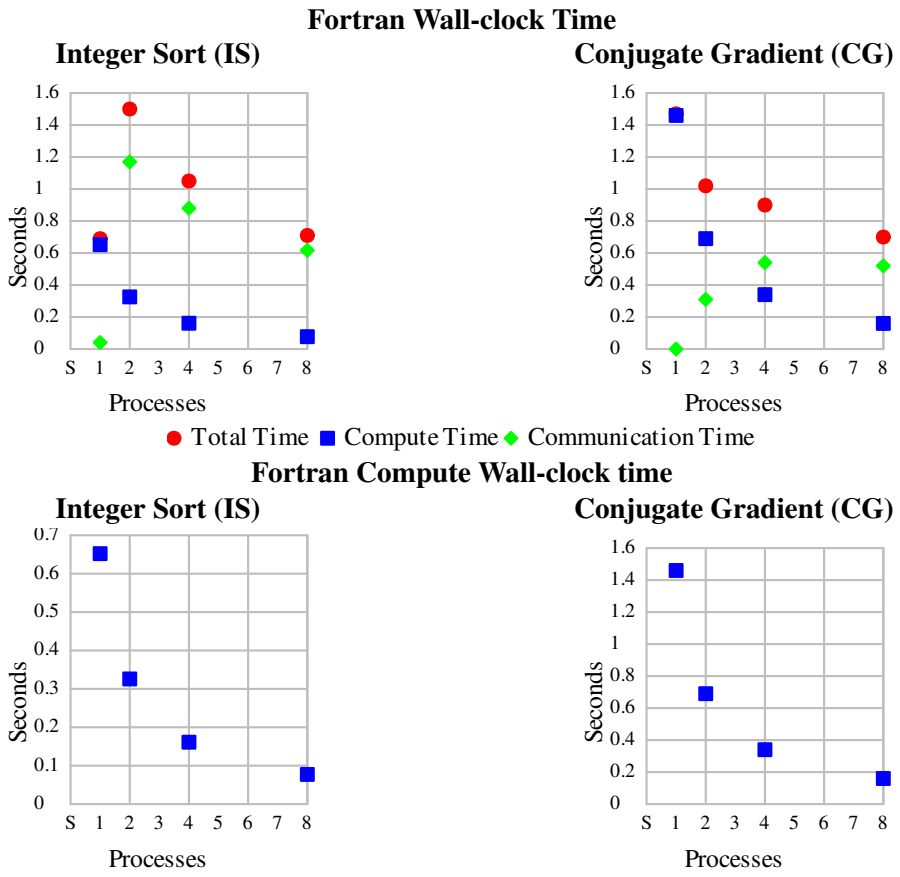

Figure 4.11: Fortran IS, CG, and MG class A results

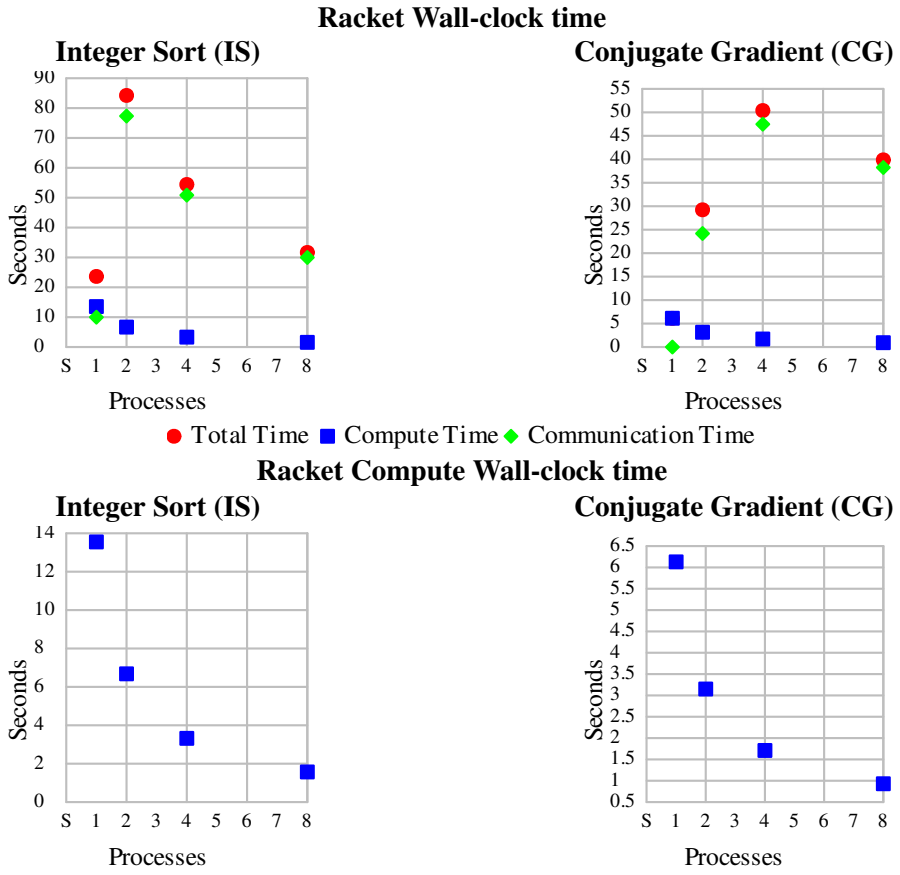

Figure 4.12: Racket IS, CG, and MG class A results

4.6 Distributed Places Complete API

Distributed places support programs whose computation may span physical machines. The design relies on machine *nodes* that perform computation. The programmer configures a new distributed system using a declarative syntax and callbacks. A node begins life with one initial place: the *message router*. After a node has been configured, its message router is activated by calling the `message-router` function. The message router listens on a TCP port for incoming connections from other nodes in the distributed system. Places can be spawned within the node by sending place-spawn request messages to the node's message router.

The distributed places implementation relies on two assumptions:

- The user's `".ssh/config"` and `".ssh/authorized_keys"` files are configured correctly to allow passwordless connection to remote hosts via public key authentication.
- Distributed places does not support the specification of ssh usernames. If a nondefault ssh username is required, the `".ssh/config"` file should be used to specifiy the username.
- All machines run the same version of Racket. Futures versions of distributed places may use the zo binary data format for serialization.

The following example, in Figure 4.13, illustrates a configuration and use of distributed places that starts a new node on the current machine and passes it a `"Hello World"` string.

```
(message-router ec ...+) → void?
  ec : (is-a?/c event-container<%>)
```

waits in an endless loop for one of many events to become ready. The `message-router` procedure constructs a `node%` instance to serve as the message router for the node. The `message-router` procedure then adds all the declared `event-container<%>`s to the `node%` and finally calls the never ending loop sync-events method, which handles events for the node.

```
(spawn-node-with-place-at
 hostname
 instance-module-path
 instance-place-function-name
 [#:listen-port port
 #:initial-message initial-message
 #:racket-path racket-path
 #:ssh-bin-path ssh-path
 #:distributed-launch-path launcher-path
 #:restart-on-exit restart-on-exit
 #:named place-name
 #:thunk thunk])
```

```
(module hello-world-example racket/base
  (require racket/place/distributed
           racket/place)

  (provide hello-world)

  (define (hello-world)
    (place ch
      (printf "hello-world received: ~a\n" (place-channel-get ch))
      (place-channel-put ch "Hello World\n")
      (printf "hello-world sent: Hello World\n")))


  (module+ main
    ; 1) spawns a node running at "localhost" and listenting on port
    ; 6344 for incomming connections.
    ; 2) connects to the node running at localhost:6344 and creates a
    ; place on that node by calling the hello-world procedure from
    ; the current module.
    ; 3) returns a remote-node% instance (node) and a
    ; remote-connection% instance (pl) for communicating with the
    ; new node and place
    (define-values (node pl)
      (spawn-node-supervise-place-at "localhost"
                                     #:listen-port 6344
                                     #:thunk #t
                                     (quote-module-path "..")
                                     'hello-world))

    ; starts a message router which adds three event-container<%>s to
    ; its list of events to handle: the node and two after-seconds
    ; event containers . Two seconds after the launch of the
    ; message-router, a  message will be sent to the pl place.  After
    ; six seconds, the program and all spawned nodes and places will
    ; terminate.
    (message-router
      node
      (after-seconds 2
        (*channel-put pl "Hello")
        (printf "message-router received: ~a\n" (*channel-get pl)))

      (after-seconds 6
        (exit 0)))))
```

Figure 4.13: Distributed Places "Hello World"

```
→ (is-a?/c remote-connection%)
 hostname : string?
 instance-module-path : module-path?
 instance-place-function-name : symbol?
 port : port-no? = DEFAULT-ROUTER-PORT
 initial-message : any = #f
 racket-path : string-path? = (racket-path)
 ssh-path : string-path? = (ssh-bin-path)
 launcher-path :  string-path?
               = (path->string distributed-launch-path)
 restart-on-exit : any/c = #f
 place-name : (or/c #f symbol?) = #f
 thunk : (or/c #f #t) = #f
```

spawns a new remote node at *hostname* with one instance place specified by the *instance-module-path* and *instance-place-function-name*. When *thunk* is `#f`, the place is created as the result of the framework calling `(dynamic-place` *instance-module-path instance-place-function-name*`)`. in the new node. When *thunk* is `#t`, the *instance-place-function-name* function should use `dynamic-place` or `place` to create and return an initial place in the new node. When the *place-name* symbol is present, a named place is created. The *place-name* symbol is used to establish later connections to the named place.

The result is a `remote-node%` instance, not a `remote-connection%`. Use get-first-place on the result to obtain a `remote-connection%`.

The *restart-on-exit* argument can be `#t` to instruct the `remote-connection%` instance to respawn the place on the remote node should it exit or terminate at any time. It can also be a procedure of zero arguments to implement the restart procedure, or it can be an object that supports a `restart` method that takes a place argument.

```
(spawn-node-supervise-place-at
 hostname
 instance-module-path
 instance-place-function-name
 [#:listen-port port
  #:initial-message initial-message
  #:racket-path racket-path
  #:ssh-bin-path ssh-path
  #:distributed-launch-path launcher-path
  #:restart-on-exit restart-on-exit
  #:named named
  #:thunk thunk])
  (is-a?/c remote-node%)
→ (is-a?/c remote-connection%)
```

```
  hostname : string?
  instance-module-path : module-path?
  instance-place-function-name : symbol?
  port : port-no? = DEFAULT-ROUTER-PORT
  initial-message : any = #f
  racket-path : string-path? = (racket-path)
  ssh-path : string-path? = (ssh-bin-path)
  launcher-path : string-path?
                = (path->string distributed-launch-path)
  restart-on-exit : any/c = #f
  named : (or/c #f string?) = #f
  thunk : (or/c #f #t) = #f
```

like `spawn-node-with-dynamic-place-at`, but the result is two values: the new `remote-node%` and its `remote-connection%` instance.

```
(spawn-remote-racket-node
 hostname
 [#:listen-port port
  #:racket-path racket-path
  #:ssh-bin-path ssh-path
  #:distributed-launch-path launcher-path]
  #:use-current-ports use-current-ports)
→ (is-a?/c remote-node%)
 hostname : string?
 port : port-no? = DEFAULT-ROUTER-PORT
 racket-path : string-path? = (racket-path)
 ssh-path : string-path? = (ssh-bin-path)
 launcher-path : string-path?
               = (path->string distributed-launch-path)
 use-current-ports : #f
```

spawns a new remote node at *hostname* and returns a `remote-node%` handle.

```
(create-place-node  hostname
                   [#:listen-port port
                    #:racket-path racket-path
                    #:ssh-bin-path ssh-path
                    #:distributed-launch-path launcher-path
                    #:use-current-ports use-current-ports])
→ (is-a?/c remote-node%)
 hostname : string?
 port : port-no? = DEFAULT-ROUTER-PORT
 racket-path : string-path? = (racket-path)
 ssh-path : string-path? = (ssh-bin-path)
 launcher-path : string-path?
               = (path->string distributed-launch-path)
```

```
  use-current-ports : boolean? = #t
```

like spawn-remote-racket-node, but the current-output-port and current-error-port are used as the standard ports for the spawned process instead of new pipe ports.

```
(supervise-place-at  remote-node
                     instance-module-path
                     instance-place-function-name
                    [#:restart-on-exit restart-on-exit
                     #:named named
                     #:thunk thunk])
 → (is-a?/c remote-connection%)
  remote-node : (is-a?/c remote-node%)
  instance-module-path : module-path?
  instance-place-function-name : symbol?
  restart-on-exit : any/c = #f
  named : (or/c #f symbol?) = #f
  thunk : (or/c #f #t) = #f
```

when *thunk* is #f, it creates a new place on *remote-node* by using dynamic-place to invoke *instance-place-function-name* from the module *instance-module-path*. When *thunk* is #t, it creates a new place at *remote-node* by executing the thunk exported as *instance-place-function-name* from the module *instance-module-path*. The function should use dynamic-place or place to create and return a place in the new node. When the place-name symbol is present, a named place is created. The place-name symbol is used to establish later connections to the named place.

```
(supervise-process-at  hostname
                       commandline-argument ...+
                      [#:listen-port port])
 → (is-a?/c remote-process%)
  hostname : string?
  commandline-argument : string?
  port : port-no? = DEFAULT-ROUTER-PORT
```

spawns an attached external process at host *hostname*.

```
(supervise-thread-at  remote-node
                      instance-module-path
                      instance-thunk-function-name
                     [#:restart-on-exit restart-on-exit])
 → (is-a?/c remote-connection%)
  remote-node : (is-a?/c remote-node%)
  instance-module-path : module-path?
  instance-thunk-function-name : symbol?
  restart-on-exit : any/c = #f
```

creates a new thread on the *remote-node* by using `dynamic-require` to invoke `instance-place-function-name` from the module *instance-module-path*.

```
(restart-every  seconds
                [#:retry retry
                 #:on-final-fail on-final-fail])
 → (is-a/c respawn-and-fire%)
  seconds : (number?)
  retry : (or/c number? #f) = #f
  on-final-fail : (or/c #f (-> any/c)) = #f
```

returns a `restarter%` instance that should be supplied to a `#:restart-on-exit` argument.

```
(every-seconds seconds-expr body ....)
```

returns a `respawn-and-fire%` instance that should be supplied to a `message-router`. The `respawn-and-fire%` instance executes *body*s once every *N* seconds, where *N* is the result of *seconds-expr*.

```
(after-seconds seconds-expr body ....)
```

returns a `after-seconds%` instance that should be supplied to a `message-router`. The `after-seconds%` instance executes the *body*s after a delay of *N* seconds from the start of the event loop, where *N* is the result of *seconds-expr*.

```
(connect-to-named-place node name)
 → (is-a?/c remote-connection%)
  node : (is-a?/c remote-node%)
  name : symbol?
```

connects to a named place on the *node* named *name* and returns a `remote-connection%` object.

```
(log-message severity msg) → void?
  severity : (or/c 'fatal 'error 'warning 'info 'debug)
  msg : string?
```

logs a message at the root node.

```
event-container<%> : interface?
```

All objects that are supplied to the `message-router` must implement the `event-container<%>` interface. The `message-router` calls the `register` method on each supplied `event-container<%>` to obtain a list of events on which the event loop should wait.

```
(send an-event-container register events) → (listof events?)
  events : (listof events?)
```

Returns the list of events inside the `event-container<%>` that should be waited on by the `message-router`.

The following classes all implement `event-container<%>` and can be supplied to a `message-router`: `spawned-process%`, `place-socket-bridge%`, `node%`, `remote-node%`, `remote-connection%`, `place% connection%`, `respawn-and-fire%`, and `after-seconds%`.

```
spawned-process% : class?
  superclass: object%
  extends: event-container<%>
```

```
(send a-spawned-process get-pid) → exact-positive-integer?
```

```
(new spawned-process%
     [cmdline-list cmdline-list]
    [[parent parent]])
→ (is-a?/c spawned-process%)
 cmdline-list : (listof (or/c string? path?))
 parent : (is-a?/c remote-node%) = #f
```

The *cmdline-list* is a list of command line arguments of type `string` and/or `path`.

The *parent* argument is a `remote-node%` instance that will be notified when the process dies via a `(send parent process-died this)` call.

```
place-socket-bridge% : class?
  superclass: object%
  extends: event-container<%>
```

```
(send a-place-socket-bridge get-sc-id)
 → exact-positive-integer?
```

```
(new place-socket-bridge%
    [pch pch]
    [sch sch]
    [id id])
→ (is-a?/c place-socket-bridge%)
 pch : place-channel?
 sch : (is-a?/c socket-connection%)
 id : exact-positive-integer?
```

The *pch* argument is a `place-channel`. Messages received on *pch* are forwarded to the socket-connection% *sch* via a `dcgm` message, e.g., `(sconn-write-flush sch (dcgm DCGM-TYPE-INTER-DCHANNEL id id msg))`. The *id* is an `exact-positive-integer` that identifies the socket-connection subchannel for this internode place connection.

```
socket-connection% : class?
  superclass: object%
  extends: event-container<%>
```

```
(new socket-connection%
   [[host host]
    [port port]
    [retry-times retry-times]
    [delay delay]
    [background-connect? background-connect?]
    [in in]
    [out out]
    [remote-node remote-node]])
→ (is-a?/c socket-connection%)
 host : (or/c string? #f) = #f
 port : (or/c port-no? #f) = #f
 retry-times : exact-nonnegative-integer? = 30
 delay : number? = 1
 background-connect? : any/c = #f
 in : (or/c input-port? #f) = #f
 out : (or/c output-port #f) = #f
 remote-node : (or/c (is-a?/c remote-node%) #f) = #f
```

When a *host* and *port* are supplied, a new tcp connection is established. If a `input-port?` and `output-port?` are supplied as *in* and *out*, the ports are used as a connection to the remote host. The *retry-times* argument specifies how many times to retry the connection attempt should it fail to connect and defaults to 30 retry attempts. Often a remote node is still booting up when a connection is attempted and the connection needs to be retried several times. The *delay* argument specifies how many seconds to wait between retry attempts. The *background-connect?* argument defaults to `#t` and specifies that the constructor should retry immediately and that connection establishment should occur in the background. Finally, the *remote-node* argument specifies the `remote-node%` instance that should be notified should the connection fail.

```
node% : class?
  superclass: object%
  extends: event-container<%>
```

The `node%` instance controls a distributed places node. It launches places and routes internode place messages in the distributed system. The `message-router` form constructs a `node%` instance under the hood. Newly spawned nodes also have a `node%` instance in their initial place that serves as the node's message router.

```
(new node% [[listen-port listen-port]]) → (is-a?/c node%)
  listen-port : tcp-listen-port? = #f
```

constructs a node% that will listen on *listen-port* for internode connections.

(send *a-node* sync-events) → void?

starts the never ending event loop for this distributed places node.

remote-node% : class?
  superclass: object%
  extends: event-container<%>

The node% instance controls a distributed places node. It launches compute places and routes internode place messages in the distributed system. This is the remote api to a distributed places node. Instances of remote-node% are returned by spawn-remote-racket-node, spawn-node-supervise-dynamic-place-at, and spawn-node-supervise-place-thunk-at.

(new remote-node%
   [[listen-port *listen-port*]
    [restart-on-exit *restart-on-exit*]])
→ (is-a?/c remote-node%)
 listen-port : tcp-listen-port? = #f
 restart-on-exit : any/c = #f

constructs a node% that will listen on *listen-port* for internode connections.

When set to true, the *restart-on-exit* parameter causes the specified node to be restarted when the ssh session spawning the node dies.

(send *a-remote-node* get-first-place)
 → (is-a?/c remote-connection%)

returns the remote-connection% object instance for the first place spawned on this node.

(send *a-remote-node* get-first-place-channel) → place-channel?

returns the communication channel for the first place spawned on this node.

(send *a-remote-node* get-log-prefix) → string?

returns (format "PLACE ~a:~a" host-name listen-port)

(send *a-remote-node* launch-place
  place-exec
  [#:restart-on-exit *restart-on-exit*
   #:one-sided-place? *one-sided-place?*])
→ (is-a?/c remote-connection%)
 place-exec : list?
 restart-on-exit : any/c = #f
 one-sided-place? : any/c = #f

launches a place on the remote node represented by this `remote-node%` instance.

The *`place-exec`* argument describes how the remote place should be launched, and it should have one of the following shapes:

- `(list 'place` *`place-module-path place-thunk`*`)`
- `(list 'dynamic-place` *`place-module-path place-func`*`)`

The difference between these two launching methods is that the `'place` version of *`place-exec`* expects a thunk to be exported by the module `place-module-path`. Executing the thunk is expected to create a new place and return a place descriptor to the newly created place. The `'dynamic-place` version of *`place-exec`* expects place-func to be a function taking a single argument, the initial channel argument, and calls `dynamic-place` on behalf of the user and creates the new place from the `place-module-path` and `place-func`.

The *`restart-on-exit`* argument is treated in the same way as for `spawn-node-with-dynamic-place-at`.

The *`one-sided-place?`* argument is an internal use argument for launching remote places from within a place using the old design pattern.

`(send` *`a-remote-node`* remote-connect *`name`*`)` → `remote-connection%`
  `name` : `string?`

connects to a named place on the remote node represented by this `remote-node%` instance.

`(send` *`a-remote-node`* send-exit`)` → `void?`

sends a message instructing the remote node represented by this `remote-node%` instance to exit immediately.

`(node-send-exit` *`remote-node%`*`)` → `void?`
  `remote-node%` : `node`

sends `node` a message telling it to exit immediately.

`(node-get-first-place` *`remote-node%`*`)`
 → `(is-a?/c remote-connection%)`
  `remote-node%` : `node`

returns the `remote-connection%` instance of the first place spawned at this node.

`(distributed-place-wait` *`remote-connection%`*`)` → `void?`
  `remote-connection%` : `place`

waits for `place` to terminate.

```
remote-connection% : class?
  superclass: object%
  extends: event-container<%>
```

The `remote-connection%` instance provides a remote api to a place running on a remote distributed places node. It launches a places or connects to a named place and routes internode place messages to the remote place.

```
(new remote-connection%
   [node node]
   [place-exec place-exec]
   [name name]
   [restart-on-exit restart-on-exit]
   [one-sided-place? one-sided-place?]
   [on-channel on-channel])
→ (is-a?/c remote-connection%)
 node : (is-a?/c remote-node%)
 place-exec : list?
 name : string?
 restart-on-exit : #f
 one-sided-place? : #f
 on-channel : #f
```

constructs a `remote-connection%` instance. The *`place-exec`* argument describes how the remote place should be launched in the same way as for launch-place in `remote-node%`. The *`restart-on-exit`* argument is treated in the same way as for `spawn-node-with-dynamic-place-at`.

The *`one-sided-place?`* argument is an internal use argument for launching remote places from within a place using the old design pattern.

See `set-on-channel!` for description of the *`on-channel`* argument.

```
(send a-remote-connection set-on-channel! callback) → void?
  callback : (-> channel msg void?)
```

installs a handler function that handles messages from the remote place. The `setup/distributed-docs` module uses this callback to handle job completion messages.

```
place% : class?
  superclass: object%
  extends: event-container<%>
```

The `place%` instance represents a place launched on a distributed places node at that node. It launches a compute places and routes internode place messages to the place.

```
(new place%
   [node node]
   [place-exec place-exec]
   [ch-id ch-id]
   [sc sc]
  [[on-place-dead on-place-dead]]) → (is-a?/c place%)
 node : (is-a?/c remote-connection%)
 place-exec : list?
 ch-id : exact-positive-integer?
 sc : (is-a?/c socket-connection%)
 on-place-dead : (-> event void?) = default-on-place-dead
```

constructs a `remote-connection%` instance. The *`place-exec`* argument describes how the remote place should be launched in the same way as for launch-place in `remote-node%`. The *`ch-id`* and *`sc`* arguments are internally used to establish routing between the remote node spawning this place and the place itself. The *`on-place-dead`* callback handles the event when the newly spawned place terminates.

```
(send a-place wait-for-die) → void?
```

blocks and waits for the subprocess representing the `remote-node%` to exit.

```
connection% : class?
  superclass: object%
  extends: event-container<%>
```

The `connection%` instance represents a connection to a named-place instance running on the current node. It routes internode place messages to the named place.

```
(new connection%
   [node node]
   [name name]
   [ch-id ch-id]
   [sc sc])       → (is-a?/c connection%)
 node : (is-a?/c remote-node%)
 name : string?
 ch-id : exact-positive-integer?
 sc : (is-a?/c socket-connection%)
```

constructs a `remote-connection%` instance. The `place-exec` argument describes how the remote place should be launched in the same way as for launch-place in `remote-node%`. The *`ch-id`* and *`sc`* arguments are internally used to establish routing between the remote node and this named-place.

```
respawn-and-fire% : class?
  superclass: object%
  extends: event-container<%>
```

The `respawn-and-fire%` instance represents a thunk that should execute every `n` seconds.

```
(new respawn-and-fire%
   [seconds seconds]
   [thunk thunk])
→ (is-a?/c respawn-and-fire%)
 seconds : (and/c real? (not/c negative?))
 thunk : (-> void?)
```

constructs a `respawn-and-fire%` instance that when placed inside a `message-router` construct causes the supplied thunk to execute every `n` seconds.

```
after-seconds% : class?
  superclass: object%
  extends: event-container<%>
```

The `after-seconds%` instance represents a thunk that should execute after `n` seconds.

```
(new after-seconds%
   [seconds seconds]
   [thunk thunk])
→ (is-a?/c after-seconds%)
 seconds : (and/c real? (not/c negative?))
 thunk : (-> void?)
```

constructs an `after-seconds%` instance that when placed inside a `message-router` construct causes the supplied thunk to execute after `n` seconds.

```
restarter% : class?
  superclass: after-seconds%
  extends: event-container<%>
```

The `restarter%` instance represents a restart strategy.

```
(new restarter%
    [seconds seconds]
   [[retry retry]
    [on-final-fail on-final-fail]])
→ (is-a?/c restarter%)
 seconds : number?
 retry : (or/c number? #f) = #f
 on-final-fail : (or/c #f (-> any/c)) = #f
```

constructs a `restarter%` instance that when supplied to a `#:restart-on-exit` argument, attempts to restart the process every *`seconds`*. The *`retry`* argument specifies how many times to attempt to restart the process before giving up. If the process stays alive for `(* 2 seconds)`, the attempted retries count is reset to `0`. The *`on-final-fail`* thunk is called when the number of retries is exceeded

```
distributed-launch-path : path?
```

contains the local path to the distributed places launcher. The distributed places launcher is the bootsrap file that launches the message router on a new node.

```
(ssh-bin-path) → string?
```

returns the path to the ssh binary on the local system in string form.
Example:

```
> (ssh-bin-path)
#<path:/usr/bin/ssh>
```

```
(racket-path) → path?
```

returns the path to the currently executing Racket binary on the local system.

```
(build-distributed-launch-path collects-path) → string?
  collects-path : path-string?
```

returns the path to the distributed places launch file. The function can take an optional argument specifying the path to the collects directory.

```
(spawn-node-at  hostname
                [#:listen-port port
                 #:racket-path racket-path
                 #:ssh-bin-path ssh-path
                 #:distributed-launch-path launcher-path])
 → channel?
  hostname : string?
  port : port-no? = DEFAULT-ROUTER-PORT
  racket-path : string-path? = (racket-path)
  ssh-path : string-path? = (ssh-bin-path)
  launcher-path :  string-path?
                 = (path->string distributed-launch-path)
```

spawns a node in the background using a Racket thread and returns a channel that becomes ready with a `remote-node%` once the node has spawned successfully

```
(spawn-nodes/join nodes-descs) → void?
  nodes-descs : list?
```

spawns a list of nodes by calling `(lambda (x) (apply keyword-apply spawn-node-at x))` for each node description in *nodes-descs* and then waits for each node to spawn.

```
(*channel-put ch msg) → void?
  ch : (or/c place-channel? async-bi-channel?
             channel? (is-a?/c remote-connection%))
  msg : any
```

sends *msg* over *ch* channel.

```
(*channel-get ch) → any
  ch : (or/c place-channel? async-bi-channel?
             channel? (is-a?/c remote-connection%))
```

returns a message received on *ch* channel.

```
(*channel? v) → boolean?
  v : any/c
```

returns `#t` if *v* is one of `place-channel?`, `async-bi-channel?`, `channel?`, or `(is-a?/c remote-connection%)`.

```
(send-new-place-channel-to-named-dest ch
                                      src-id
                                      dest-list)
 → place-channel?
  ch : *channel?
  src-id : any
  dest-list : (listof string? port-no? string?)
```

creates and returns a new place channel connection to a named place at *dest-list*. The *dest-list* argument is a list of a remote-hostname remote-port and named-place name. The channel *ch* should be a connection to a `message-router`.

```
(mr-spawn-remote-node  mrch
                       host
                      [#:listen-port listen-port
                       #:solo solo])              → void?
  mrch : *channel?
  host : string?
  listen-port : port-no? = DEFAULT-ROUTER-PORT
  solo : boolean? = #f
```

sends a message to a message router over *mrch* channel asking the message router to spawn a new node at *host* listening on port *listen-port*. If the #:solo keyword argument is supplied, the new node is not folded into the complete network with other nodes in the distributed system.

```
(mr-supervise-named-dynamic-place-at mrch
                                     dest
                                     name
                                     path
                                     func) → void?
  mrch : *channel?
  dest : (listof string? port-no?)
  name : string?
  path : string?
  func : symbol?
```

sends a message to a message router over *mrch* channel asking the message router to spawn a named place at *dest* named *name*. The place is spawned at the remote node by calling a dynamic place with module-path *path* and function *func*. The *dest* parameter should be a list of remote-hostname and remote-port.

```
(mr-connect-to mrch dest name) → void?
  mrch : *channel?
  dest : (listof string? port-no?)
  name : string?
```

sends a message to a message router over *mrch* channel asking the message router to create a new connection to the named place named *name* at *dest*. The *dest* parameter should be a list of remote-hostname and remote-port.

```
(start-message-router/thread [#:listen-port listen-port
                              #:nodes nodes])
 → thread? channel?
  listen-port : port-no? = DEFAULT-ROUTER-PORT
  nodes : list? = null
```

starts a message router in a Racket thread connected to *nodes*, listening on port *listen-port*, and returns a channel? connection to the message router.

```
(port-no? no) → boolean?
  no : (and/c exact-nonnegative-integer?  (integer-in 0 65535))
```

returns #t if *no* is an exact-nonnegative-integer? between 0 and 65535.

```
DEFAULT-ROUTER-PORT : port-no?
```

the default port for a distributed places message router.

```
named-place-typed-channel% : class?
  superclass: object%
```

```
(new named-place-typed-channel% [ch ch])
 → (is-a?/c named-place-typed-channel%)
  ch : place-channel?
```

The *ch* argument is a `place-channel`.

```
(send a-named-place-typed-channel get type) → any
  type : symbol?
```

returns the first message received on `ch` that has the type *type*. Messages are lists and their type is the first item of the list, which should be a `symbol?`. Messages of other types that are received are queued for later `get` requests.

```
(tc-get type ch) → void?
  type : symbol?
  ch : place-channel?
```

gets a message of type *type* from the `named-place-typed-channel%` *ch*.

```
(write-flush datum port) → void?
  datum : any
  port : port?
```

writes *datum* to *port* and then flushes *port*.

```
(printf/f format args ...) → void?
  format : string?
  args : any
```

calls `printf` followed by a call to `flush-output`.

```
(displayln/f item) → void?
  item : any
```

calls `displayln` followed by a call to `flush-output`.

Example:

```
> (write-flush "Hello World" (current-output-port))
"Hello World"
```

4.7 Conclusion

Building distributed places as a language extension allows the compact and clean construction of higher-level abstractions such as RPC, MPI, map reduce, and nested data parallelism. Distributed places programs are more compact and easier to write than traditional C MPI programs. A Racket MPI implementation of parallel k-means was written with distributed places using less than half the lines of code of the original C and MPI version. With distributed places, messages can be heterogeneous and serialization is handled automatically by the language.

In addition to distributed parallel computing, Racket has many features that make it a great coordination and control language. Rackets provides a rich FFI (foreign function interface) for invoking legacy C code. Racket also includes extensive process exec capabilities for launching external programs and communicating with them over standard IO pipes. Racket's FFI, process exec capabilities, and distributed places gives programmers a powerful distributed coordination and workflow language.

With distributed places, programmers can quickly develop parallel and distributed solutions to everyday problems. Developers can also build new distributed computing frameworks using distributed places as a common foundation. Distributed places extension of places augments the Racket programmer's toolbox and provides a road map other language implementers to follow.

CHAPTER 5

FUTURE WORK

The places and distributed places features of Racket empowers programmers to utilize multi-core chips and distributed machines. The places design also provides a roadmp language designers can follow to add parallelism to existing, dynamic languages that were primarily created to express sequential programs.

Places are essentially copies of the Racket VM, all running within the same operating system process. Early in the development of places, the initial plan was to enable sharing among places of read-only resources such as module definitions, byte code, and jitted code. Because these objects would be shared between places, they would have to be allocated from the master GC realm. Master GC allocations, however, trigger master GC collections, which require synchronized collection across all places. In a space-time tradeoff, the decision was made to consume space with duplicate module definitions, byte code, and jitted code. The alternative was to consume time with more frequent master GC collections. Future work should examine what could be done to make collection of the master GC realm more efficient. Future work should also further explore the space-time tradeoff of resource sharing vs master GC collection.

Another possible area of future work is the construction of application specific APIs for building object graphs in the shared, master GC realm. Such APIs would have to ensure that references are not constructed from the master GC realm to place local objects. Such references violate the invariant that allows places to independently collect their local heaps. The key would be to enforce copy semantics whenever a place-local object is attached to the master GC object graph. There are plenty of parallel graph algorithms that could benefit from such a shared-object graph API.

Future work should add support for serialized closures in place channel messages. In Racket, closures are implemented in the C runtime of the language. Traversing and serializing a closed over object graph in C requires a significant amount of engineering work. Closures do not report their total size and communication costs. The total cost of closure serialization is not known until a serialization attempt is complete. A closure may close over a significant portion of the heap or even

the entire heap, making serialization prohibitively expensive. Introspection and memory accounting of closure sizes may be an interesting future topic of research.

There are many objects types which are not serializable. Workarounds for these unserializable types need to be designed and implemented. For file port objects, a workaround may be to proxy the object across place channels. Other objects, such as continuations, represent more difficult workaround problems that future reseach should address. Some objects may best serialize to a serialization-not-possible marker. Finally, remaining failures to serialize may best be represented as an exception.

Finally, distributed places could be improved to automatically deploy the Racket runtime on new remote nodes. This improvement would free users from having to ensure that Racket is installed on remote nodes before launching remote places. Distributed places could also be modified to load module byte code over place channels from parent places. This modification would alleviate the user from being responsible for distributing code to remote nodes.

CHAPTER 6

CONCLUSION

Places and *Distributed Places* demonstrate that existing dynamic-language virtual machines, such as Racket, can be transformed into effective parallel and distributed computing platforms by adding core parallel primitives. Dynamic-language virtual machines have evolved over tens of years into large sequential programs with lots of state mutation. The obvious parallelization technique, adding shared-memory threads and locks, fails to safely parallelize these virtual machines.

The places' design specifies the separation of the virtual machine's shared state into isolated place-local copies. Each place is an instance of the Racket VM with its own private set of place-local variables. All places spawned by a Racket process are represented as window threads or pthreads inside the same OS process. Since places share a common address space inside the same OS process, message passing is as simple as copying the message contents and assigning ownership of the newly copied message to the destination place.

The places' programming model does not allow memory references to be shared between places. This invariant permits each place to garbage collect its memory independently of other running places. The Racket garbage collector had to be modified to accommodate places. While the quantity of modifications was significant, modifying the existing garbage collector was much easier than the alternative of writing a new garbage collector specifically for places.

Distributed places augments the syntax and implementation of places to enable spawning of Racket processes and places on remote machines. Once a distributed place is created, the usage of the new distributed place is no different than a local Racket place. Place channel messages are transparently sent across TCP sockets to remote distributed places without any intervention by the programmer.

The abstraction power of programming language extension enables the construction of a variety of parallel constructs on the foundation of places' and distributed places' primitives. MPI-like frameworks have been built for both places and distributed places. Example implementations of RPC, map reduce, and nested data parallelism exist for distributed places.

Places and distributed places enlarge the Racket programmer's toolbox with parallel and distributed programming capabilities. Places and distributed places provide a core set of parallel primitives that serve as a foundation for higher-level frameworks to build on. Finally, places and distributed places provide a strategy for adding parallelism and distributed processing to existing dynamic language runtimes.

REFERENCES

[1] Apache Software Foundation. Hadoop. , 2012. http://hadoop.apache.org

[2] Godmar Back. Isolation, Resource Management and Sharing in the KaffeOS Java Runtime System. PhD dissertation, University of Utah, 2002.

[3] Godmar Back and Wilson C. Hsieh. The KaffeOS Java Runtime System. ACM Transactions on Programming Languages and Systems (TOPLAS), 2005. http://doi.acm.org/10.1145/1075382.1075383

[4] D. Bailey, J. Barton, T. Lasinski, and H. Simon. The NAS Parallel Benchmarks. NAS Technical Report RNR-91-002, 1991.

[5] David Beazley. Understanding the Python GIL. PyCon 2010, 2010.

[6] Artur Bergman. threads::shared. http://perldoc.perl.org/threads/shared.html, 2012. http://perldoc.perl.org/threads/shared.html

[7] Artur Bergman. threads. http://perldoc.perl.org/threads.html, 2012. http://perldoc.perl.org/threads.html

[8] Guy E. Blelloch. Programming Parallel Algorithms. Communications of the ACM, 1996.

[9] Henry Cejtin, Suresh Jagannathan, and Richard Kelsey. Higher-Order Distributed Objects. ACM Transactions on Programming Languages and Systems (TOPLAS), 1995.

[10] Phillippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarker. X10: An Object-Oriented Approach to Non-Uniform Cluster Computing. In *Proc. ACM Intl. Conf. on Object-Oriented Programming, Systems, Languages, and Applications*, 2005.

[11] Nicholas Clark. Thread. http://search.cpan.org/∼nwclark/perl-5.8.9/lib/Thread.pm, 2012. http://search.cpan.org/~nwclark/perl-5.8.9/lib/Thread.pm

[12] Katherine E. Coons, Sebastian Burckhardt, and Madanlal Musuvathi. GAMBIT: Effective Unit Testing for Concurrency Libraries. In *Proc. ACM Symp. Principles and Practice of Parallel Programming*, 2010.

[13] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. OSDI'04: Sixth Symposium on Operating System Design and Implementation, 2004.

[14] Damien Doligez and Xavier Leroy. A Concurrent, Generational Garbage Collector for a Multithreaded Implementation of ML. In *Proc. ACM Symp. Principles of Programming Languages*, 1993.

[15] Jeff Epstein, Andrew P. Black, and Simon Peyton-Jones. Haskell for the Cloud. In Proceedings of the 4th ACM symposium on Haskell (Haskell '11), 2011.

[16] Jeffrey Epstein. Functional programming for the data centre. MS thesis, University of Cambridge, 2011.

[17] Matthew Flatt and Robert Bruce Findler. Kill-Safe Synchronization Abstractions. In *Proc. ACM Conf. Programming Language Design and Implementation*, 2004.

[18] Matthew Flatt, Robert Bruce Findler, Shriram Krishnamurthi, and Matthias Felleisen. Programming Languages as Operating Systems (or, Revenge of the Son of the Lisp Machine). In *Proc. ACM Intl. Conf. Functional Programming*, 1999.

[19] Matthew Fluet, Mike Rainey, John Reppy, and Adam Shaw. Implicitly-threaded parallelism in Manticore. In *Proc. ACM Intl. Conf. Functional Programming*, 2008.

[20] Matthew Fuchs. Dreme: for Life in the Net. PhD dissertation, New York University, 1995.

[21] Guillaume Germain, Marc Feeley, and Stefan Monnier. Concurrency Oriented Programming in Termite Scheme. In *Proc. Scheme and Functional Programming*, 2006.

[22] Guy E. Blelloch, Jonathan C. Hardwick, Siddhartha Chatterjee, Jay Sipelstein, and Marco Zagha. Implementation of a portable nested data-parallel lang. In Proceedings of the fourth ACM SIGPLAN symposium on Principles and practice of parallel programming (PPOPP '93), 1993.

[23] Maurice Herlihy and J. Eliot B. Moss. Transactional Memory: Architectural Support for Lock-free Data Structures. In Proceedings of the 20th Annual International Symposium on Computer Architecture, 1993.

[24] Carl Hewitt, Peter Bishop, and Richard Steiger. A Universal Modular ACTOR Formalism for Artificial Intelligence. In Proceedings of the 3rd International Joint Conference on Artificial Intelligence (IJCAI'73), 1973.

[25] P. Hilfinger, D. Bonachea, K. Datta, D. Gay, S. Graham, B. Liblit, G. Pike, J. Su, and K. Yelick. Titanium Language Reference Manual. U.C. Berkeley Tech Report UCB/EECS-2005-15, 2005.

[26] Michael Isard, Mihai Budiur, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks. European Conference on Computer Systems (EuroSys), 2007.

[27] Patrick Maier, Phil Trinder, and Has-Wolgang Loidl. High-level Distributed-Memory Parallel Haskell in Haskell. Symposium on Implementation and Application of Functional Languages, 2011.

[28] Simon Marlow, Tim Harris, Roshan P. James, and Simon Peyton Jones. Parallel Generational-copying Garbage Collection with a Block-structured Heap. In *Proc. Intl. Symp. on Memory Management*, 2008.

[29] Simon Marlow, Simon Peyton Jones, and Satnam Singh. Runtime Support for Multicore haskell. In *Proc. ACM Intl. Conf. Functional Programming*, 2009.

[30] Message Passing Interface Forum. MPI-2: Extensions to the Message-Passing Interface. http://www.mpi-forum.org/docs/mpi2-report.pdf, 2003. `http://www.mpi-forum.org/docs/mpi2-report.pdf`

[31] MPICH. MPICH. http://www.mcs.anl.gov/mpich2, 2013.

[32] George C. Necula, Scott McPeak, Shree Prakash Rahul, and Westley Weimer. CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In *Proc. Intl. Conf. Compiler Construction*, pp. 213–228, 2002.

[33] OpenMP Architecture Review Board. OpenMP Application Program Interface Version 3.0. , 2008.

[34] Python Software Foundation. Python design note on threads. http://www.python.org/doc/faq/library/#can-t-we-get-rid-of-the-global-interpreter-lock, 2008.

[35] Python Software Foundation. multiprocessing — Process-based "threading" interface. http://docs.python.org/release/2.6.6/library/multiprocessing.html#module-multiprocessing, 2011.

[36] Ravi Rajwar and James R. Goodman. Transactional Lock-Free Execution of Lock-Based Programs. Proceedings of the Tenth Symposium on Architectural Support for Programming Languages and Operating Systems, 2002.

[37] John H. Reppy. Concurrent Programming in ML. Cambridge University Press, 1999.

[38] Harvey Richardson. High Performance Fortran: History, Overview and Current Developments. Thinking Machines Corporation TMC-261, 1996.

[39] M. C. Rinard and M. S. Lam. The Design, Implementation, and Evaluation of Jade. *ACM Transactions on Programming Languages and Systems* 20(1), pp. 1–63, 1998.

[40] Konstantinos Sagonas and Jesper Wilhelmsson. Efficient Memory Management for Concurrent Programs that use Message Passing. *Science of Computer Programming* 62(2), pp. 98–121, 2006.

[41] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A Dynamic Data Race Detector for Multithreaded Programs. *Transactions on Computer Systems* 15(4), pp. 391–411, 1997.

[42] Werner Schuster. Future of the Threading and Garbage Collection in Ruby - Interview with Koichi SasadaThread State and the Global Interpreter Lock. http://www.infoq.com/news/2009/07/future-ruby-gc-gvl-gil, 2009.

[43] Alex Schwendner. Distributed Functional Programming in Scheme. MS thesis, Massachusetts Institute of Technology, 2010. `http://groups.csail.mit.edu/commit/papers/2010/alexrs-meng-thesis.pdf`

[44] Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. x86-TSO: A Rigorous and Usable Programmer's Model for x86 Multiprocessors. *Communications of the ACM* 53(7), pp. 89–97, 2010.

[45] James Swaine, Kevin Tew, Peter Dinda, Robert Bruce Findler, and Matthew Flatt. Back to the futures: Incremental Parallelization of Existing Sequential Runtime Systems. In *Proc. ACM Intl. Conf. on Object-Oriented Programming, Systems, Languages, and Applications*, 2010.

[46] Unladen Swallow. Unladen Swallow: A faster implementation of Python. http://code.google.com/p/unladen-swallow/, 2010.

[47] Kevin Tew, James Swaine, Matthew Flatt, Robert Bruce Findler, and Peter Dinda. Places: Adding Message-Passing Parallelism to Racket. Dynamic Language Symposium 2011, 2011.

[48] Typesafe Inc. Akka. http://akka.io, 2012.

[49] UPC Languge Specification v1.2. Technical Report LBNL-59208, Berkley National Lab, 2005.

[50] Adam Wick and Matthew Flatt. Memory Accounting without Partitions. In *Proc. Intl. Symp. on Memory Management*, 2004.

[51] Katherine Yelick, Dan Bonachea, Wei-Yu Chen, Phillip Colella, Kaushik Datta, Jason Duell, Susan L. Graham, Paul Hargrove, Paul Hilfinger, Parry Husbands, Costin Iancu, Amir Kamil, Rajesh Nishtala, Jimmy Su, Michael Welcome, and Tong Wen. Productivity and Performance using Partitioned Global Address Space Languages. In Proceedings of the 2007 International Workshop on Parallel Symbolic Computation (PASCO '07). ACM, 2007.